Deep Learning (2)

- Deep Convolutional Neural Network

Yueming Wang

2016. 12. 6

problems in multi-layer neural networks

- **Fully Connected Networks**

In the multi-layer neural networks, we had made full connection between all the hidden units to all the input units. For small images (28x28 in the MNIST dataset), it was computationally feasible to learn features on the entire image. However, for larger images, e.g. 96x96, we have about 10^4 input units. If we want to learn 100 features, we would have 10^6 parameters. The feed-forward and back propagation computations would be a problem.

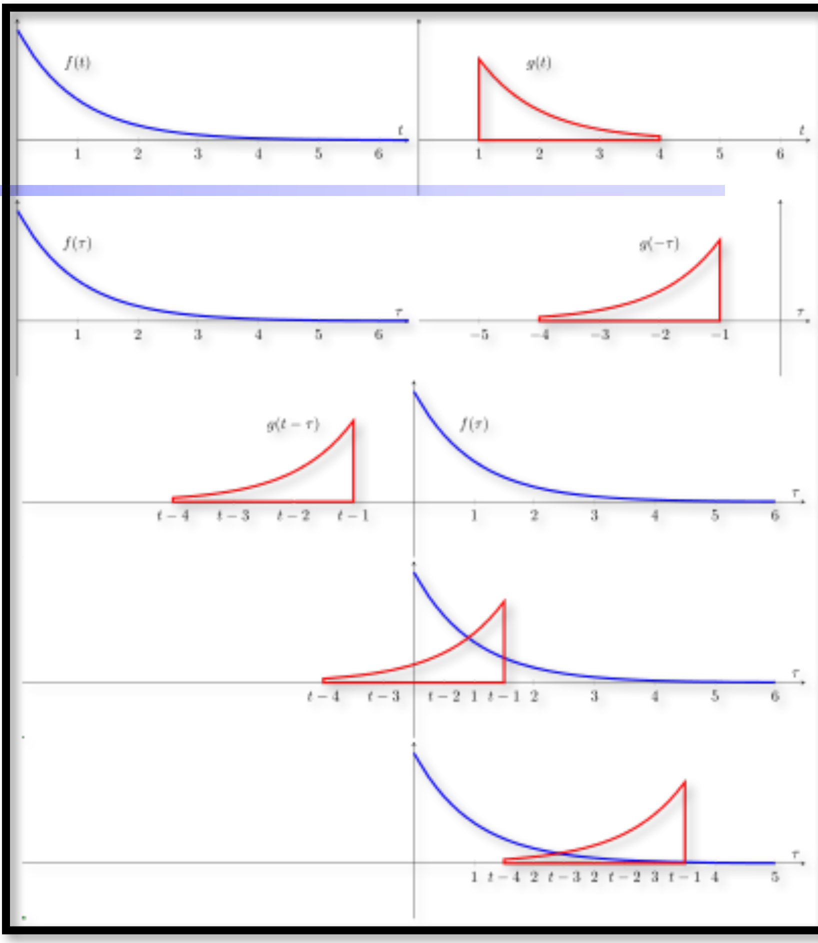
- **Locally Connected Networks**

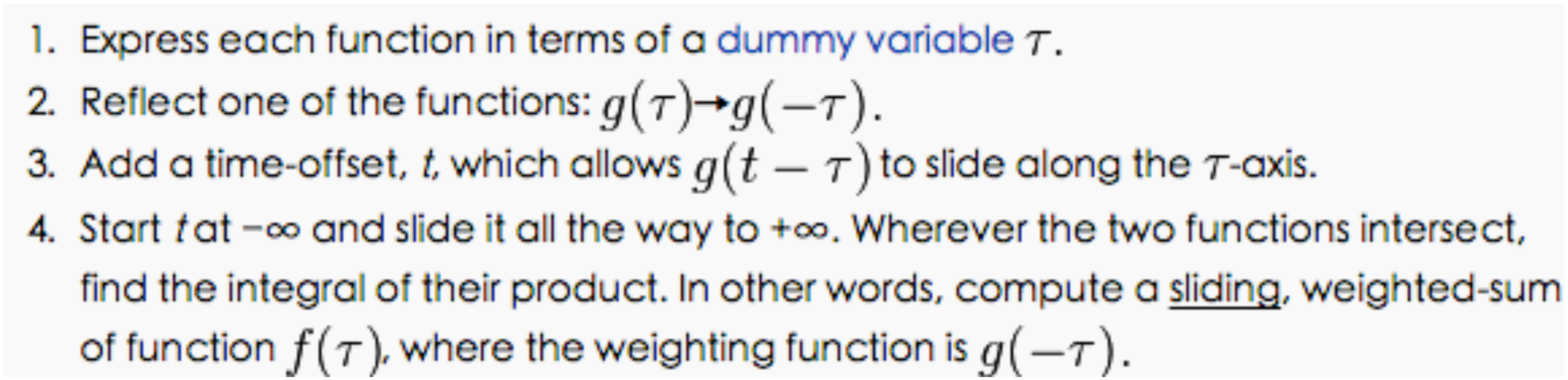
One simple solution to this problem is to restrict the connections between the hidden units and the input units, allowing each hidden unit to connect to only a small subset of the input units. Specifically, each hidden unit will connect to only a small contiguous region of pixels in the input.

1D Convolutions

Definition:

$$(f * g)(t) \overset{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau)\, g(t-\tau)\, d\tau$$





1. Express each function in terms of a dummy variable τ.
2. Reflect one of the functions: $g(\tau) \rightarrow g(-\tau)$.
3. Add a time-offset, *t*, which allows $g(t-\tau)$ to slide along the τ-axis.
4. Start *t* at $-\infty$ and slide it all the way to $+\infty$. Wherever the two functions intersect, find the integral of their product. In other words, compute a sliding, weighted-sum of function $f(\tau)$, where the weighting function is $g(-\tau)$.

1D Convolutions (discrete form)

Definition:

$$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau)\, g(t-\tau)\, d\tau$$

Discrete form:

$$(f * g)[n] \stackrel{\text{def}}{=} \sum_{m=-\infty}^{\infty} f[m]\, g[n-m]$$
$$= \sum_{m=-\infty}^{\infty} f[n-m]\, g[m].$$

If g(x) is defined in [-M, M],

$$(f * g)[n] = \sum_{m=-M}^{M} f[n-m] g[m].$$

Example:

Suppose f = [1, 2, 3, 4, 5], g = [1, 2, 3]. The computation of f * g is,

1. Flipping g to g' = [3, 2, 1], let the length be m_g;
2. Padding zeros before the first and after the last element in f:

   f' = [0, 0, 1, 2, 3, 4, 5, 0, 0], let the length be m_f;
3. Sliding g in f', for each position n (0 =< n < m_f - m_g + 1), compute sum_i (g'(i) * f(n+i)) (0 <= i < m_g)

So, f * g = [1, 4, 10, 16, 22, 22, 15], without padding, f * g = [10, 16, 22]

Extending to 2D Convolutions

f =

1	2	3
4	5	6
7	8	9

mxn

g =

1	2
3	4

lxp

f' =

0	0	0	0	0
0	1	2	3	0
0	4	5	6	0
0	7	8	9	0
0	0	0	0	0

(m+2(l-1))x(n+2(p-1))

Flip w.r.t. x axis and then y axis,

g' =

4	3
2	1

f*g =

1	4	7	6
7	23	33	24
19	53	65	42
21	52	59	36

(m+l-1)x(n+p-1)

f*g =

23	33
53	65

(m-l+1)x(n-p+1)

without padding

Note the double flip!

2D Convolutions in Scipy

f =

1	2	3
4	5	6
7	8	9

mxn

g =

1	2
3	4

lxp

f*g =

1	4	7	6
7	23	33	24
19	53	65	42
21	52	59	36

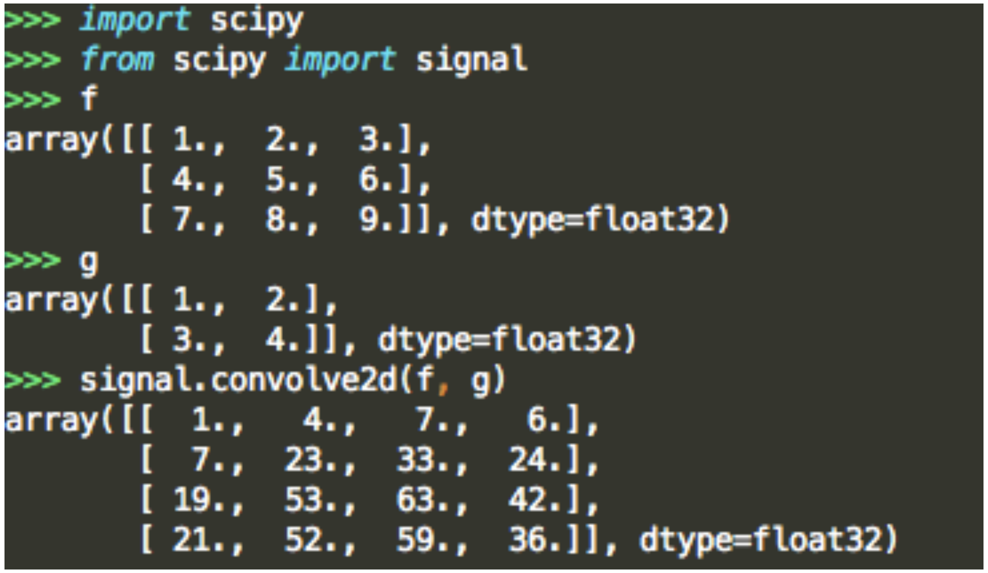
(m+l-1)x(n+p-1)

f*g =

23	33
53	65

(m-l+1)x(n-p+1)

without padding



```
>>> import scipy
>>> from scipy import signal
>>> f
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
       [ 7.,  8.,  9.]], dtype=float32)
>>> g
array([[ 1.,  2.],
       [ 3.,  4.]], dtype=float32)
>>> signal.convolve2d(f, g)
array([[  1.,   4.,   7.,   6.],
       [  7.,  23.,  33.,  24.],
       [ 19.,  53.,  63.,  42.],
       [ 21.,  52.,  59.,  36.]], dtype=float32)
```

Cross correlation

Definition:

$$(f \circ g)(\tau) \overset{\mathrm{def}}{=} \int_{-\infty}^{\infty} f^*(t)\ g(t+\tau)\,dt,$$

The cross-correlation of functions f(t) and g(t) is equivalent to the convolution of f*(−t) and g(t)

$$f \circ g = f^*(-t) * g.$$

f =

1	2	3
4	5	6
7	8	9

mxn

g =

1	2
3	4

lxp

$f \circ g$ =

4	11	18	9
18	37	47	21
36	67	77	33
14	23	26	9

$f \circ g$ =

37	47
67	77

without padding

So f ★ g = cov2(f, rot90(g, 2))!!

Cross correlation using scipy and numpy

f =

1	2	3
4	5	6
7	8	9

mxn

g =

1	2
3	4

lxp

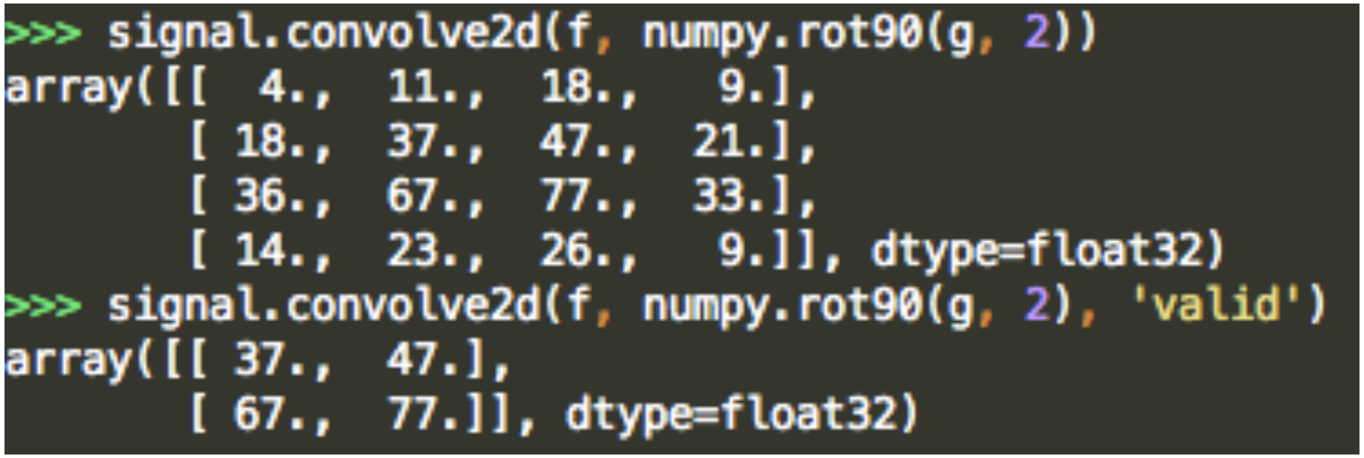
$f \circ g =$

4	11	18	9
18	37	47	21
36	67	77	33
14	23	26	9

$f \circ g =$

37	47
67	77

without padding

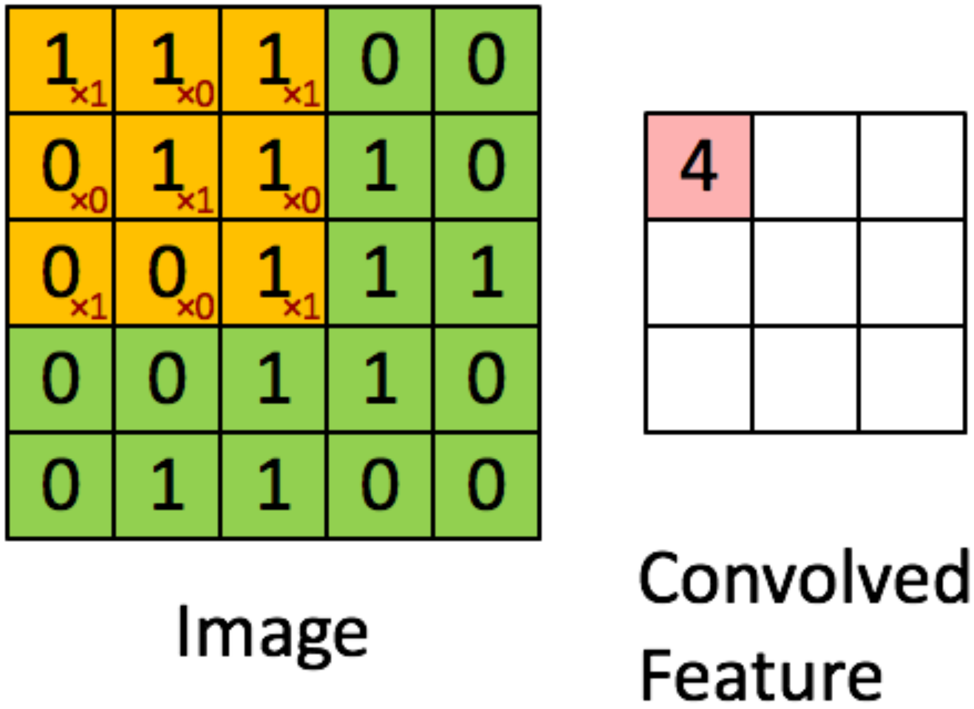


```
>>> signal.convolve2d(f, numpy.rot90(g, 2))
array([[  4.,  11.,  18.,   9.],
       [ 18.,  37.,  47.,  21.],
       [ 36.,  67.,  77.,  33.],
       [ 14.,  23.,  26.,   9.]], dtype=float32)
>>> signal.convolve2d(f, numpy.rot90(g, 2), 'valid')
array([[ 37.,  47.],
       [ 67.,  77.]], dtype=float32)
```

Cross correlation (Convolutions)

Natural images have the property of being "'stationary"', meaning that the statistics of one part of the image are the same as any other part. This suggests that the features that we learn at one part of the image can also be applied to other parts of the image, and we can use the same features at all locations.

More precisely, having learned features over small (say 8x8) patches sampled randomly from the larger image, we can then apply this learned 8x8 feature detector anywhere in the image. Specifically, we can take the learned 8x8 features and "'convolve"' them with the larger image, thus obtaining a different feature activation value at each location in the image.



How to obtain the 8x8 feature? autoencoder

Cross correlation (Convolutions)

For a 96x96 image, if we use a 8x8 patch and slide the patch anywhere in the image, we will have,

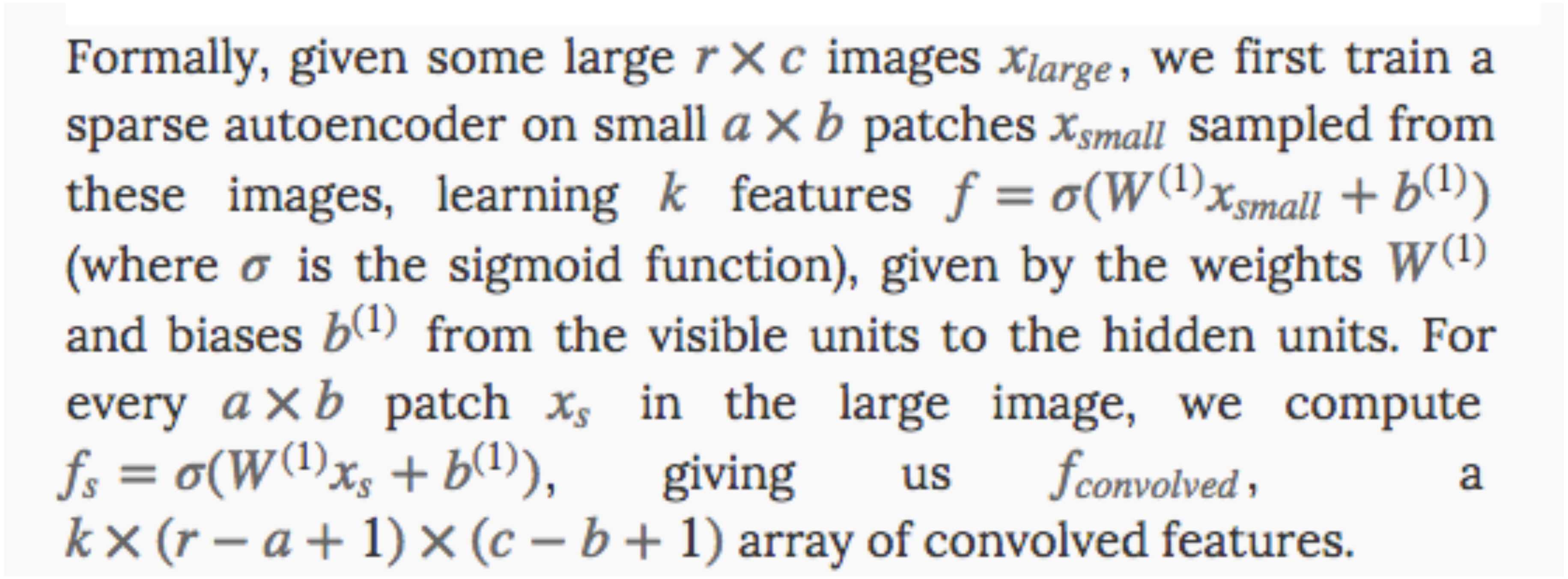
- 89x89 hidden layer nodes
- But only 8x8 parameters

If we use 100 different 8x8 patches and do the same operation, we will have,

- 89x89x100 hidden layer nodes
- But only 100x8x8 parameters

Compared with 96x96x100 parameters in a 100 hidden layer nodes in the full connection.

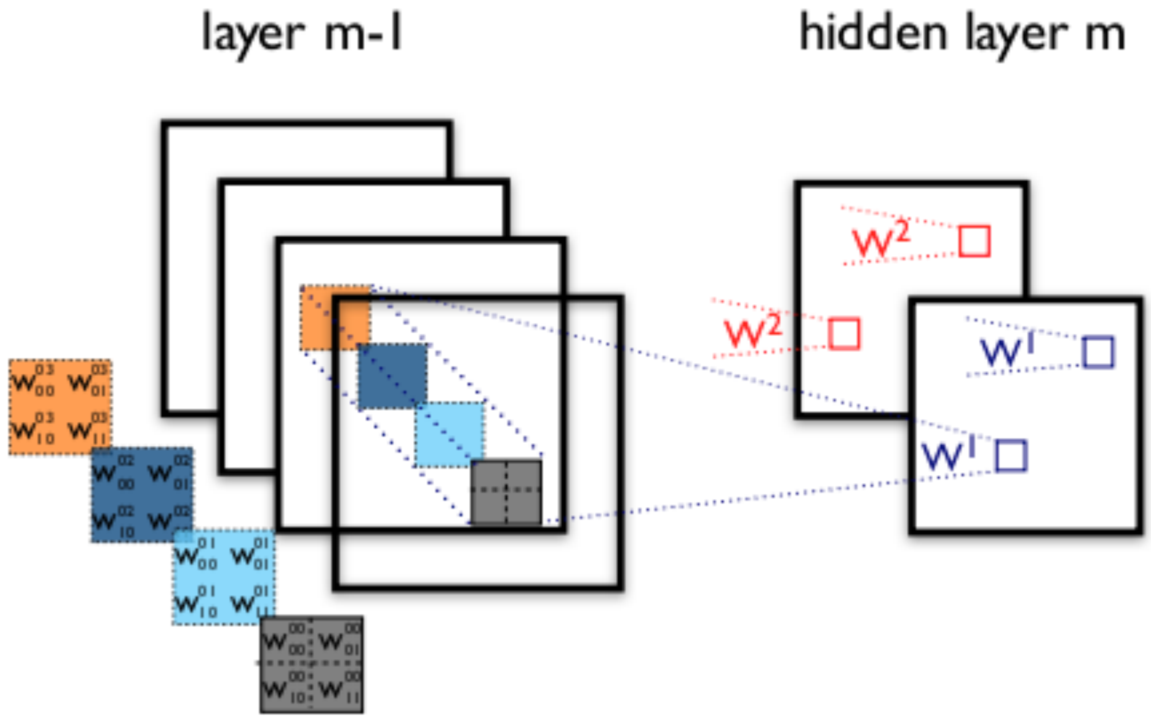
Cross correlation (Convolutions)



Formally, given some large $r \times c$ images x_{large}, we first train a sparse autoencoder on small $a \times b$ patches x_{small} sampled from these images, learning k features $f = \sigma(W^{(1)} x_{small} + b^{(1)})$ (where σ is the sigmoid function), given by the weights $W^{(1)}$ and biases $b^{(1)}$ from the visible units to the hidden units. For every $a \times b$ patch x_s in the large image, we compute $f_s = \sigma(W^{(1)} x_s + b^{(1)})$, giving us $f_{convolved}$, a $k \times (r - a + 1) \times (c - b + 1)$ array of convolved features.

Cross correlation (Convolutions)

To form a richer representation of the data, each hidden layer is composed of multiple feature maps,

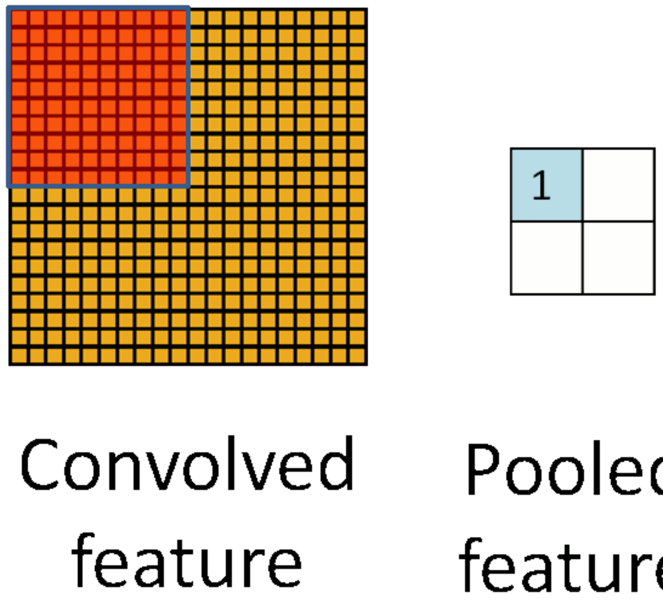


layer m-1
hidden layer m
W^2
W^2
W^1
W^1
W^{03}_{00} W^{03}_{01} W^{03}_{10} W^{03}_{11}
W^{02}_{00} W^{02}_{01} W^{02}_{10} W^{02}_{11}
W^{01}_{00} W^{01}_{01} W^{01}_{10} W^{01}_{11}
W^{00}_{00} W^{00}_{01} W^{00}_{10} W^{00}_{11}

Pooling

one could use all the extracted features with a classifier such as a softmax classifier, but this can be computationally challenging. Consider 96x96 images, and suppose we have 400 different 8x8 features. Each convolution results in an output of size (96−8+1)∗(96−8+1)=7921, and in total results in a vector of 89^2∗400=3,168,400 features per example. Learning a classifier with inputs having 3+ million features can be unfeasible.

To address this, first recall that features that are useful in one region are also likely to be useful for other regions. Thus, to describe a large image, one natural approach is to aggregate statistics of these features at various locations. For example, compute the mean (or max) value of a particular feature over a region of the image. These summary statistics are much lower in dimension and can also improve results. We aggregation operation is called it “pooling”', or sometimes ”‘mean pooling”' or ”‘max pooling”.

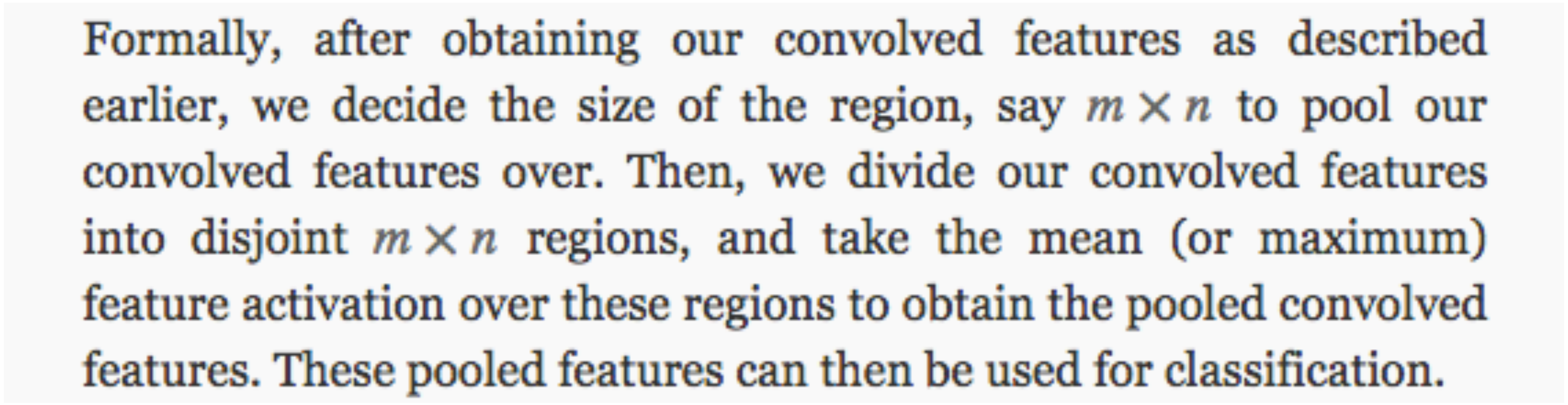


Pooling

- Translation invariant

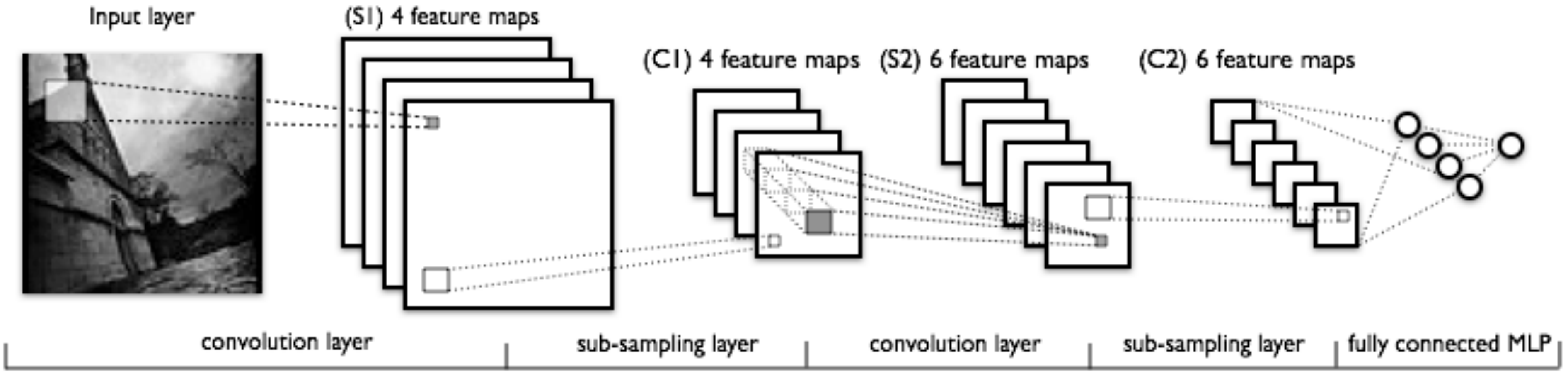
If one chooses the pooling regions to be contiguous areas in the image and only pools features generated from the same con kernel, the pooling results will then be translation invariant. This means if an image has small translations, the pooled feature can be same.

- Formal description


Formally, after obtaining our convolved features as described earlier, we decide the size of the region, say $m \times n$ to pool our convolved features over. Then, we divide our convolved features into disjoint $m \times n$ regions, and take the mean (or maximum) feature activation over these regions to obtain the pooled convolved features. These pooled features can then be used for classification.

Convolutional neural network (overview)

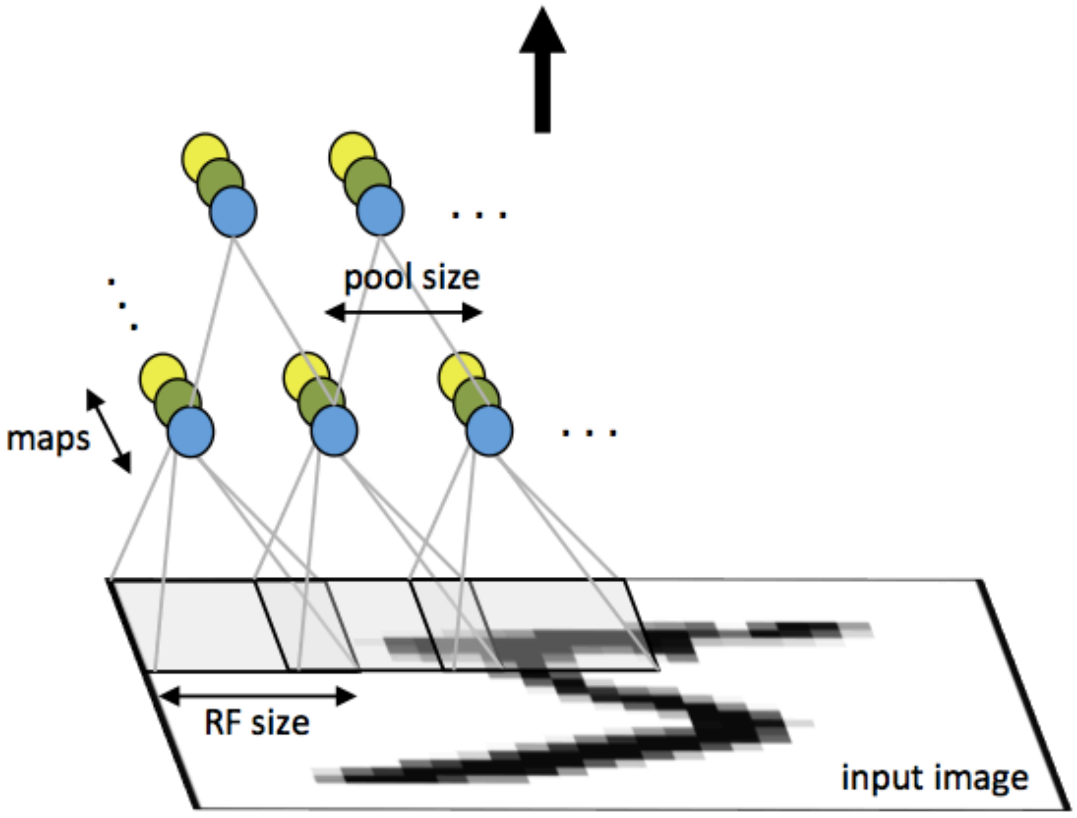
- One or more convolutional layers
- Perhaps a few subsampling step (pooling)
- One or more fully connected layers (MLP)
- Advantages:
  - 2D local structure of an image (local connections)
  - Pooling for translation invariant
  - Fewer parameters than MLP



Architecture details of CNN

- The input: a m x m x r image, where m is the height and width of the image and r is the number of channels
- a number of convolutional and subsampling layers optionally followed by fully connected layers

- Each layer has k filters (kernels) of size nxnxq
- K feature maps of size m-n+1
- Each map is subsampled with mean or max pooling over pxp regions, 2<=p<=5
- Either before or after the pooling layer, there is an additive bias and sigmoidal nonlinearity
- After the convolutional layers, there may be any number of fully connected layers



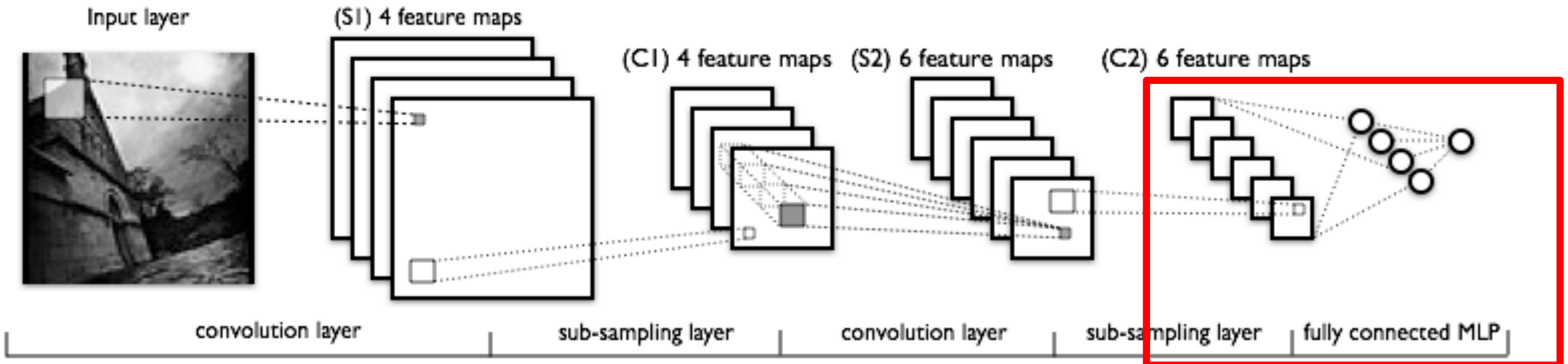
Mathematics of back propagation in CNN(1)

Log-likelihood:

$$J(W,b;x,y) = \sum_{i=1}^{S_{n_l}} 1(y=i)\ln p(y=i \mid x;W,b)$$

Output layer:

$$\delta_i^{(n_l)} = \frac{\partial J}{\partial z_i^{(n_l)}} = 1(y=i) - p(y=i)$$



Mathematics of back propagation in CNN (2)

For fully connected layers, according to the full derivative rule:

$$f = f(u, v, \ldots,), u = u(t), v = v(t), \ldots$$

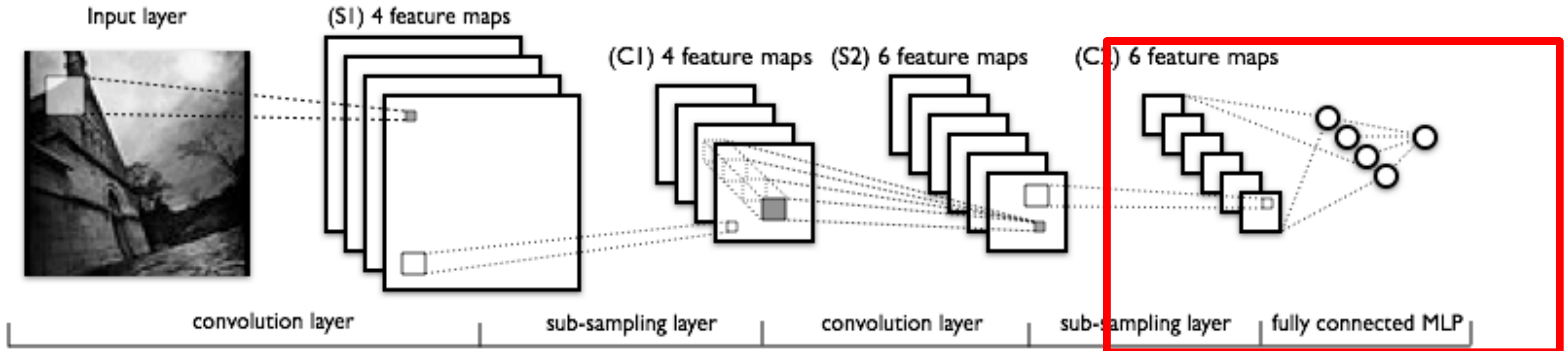
$$\frac{df}{dt} = \frac{\partial f}{\partial u}\frac{du}{dt} + \frac{\partial f}{\partial v}\frac{dv}{dt} + \ldots$$

So,

$$\frac{\partial J}{\partial z^{(l)}} = \sum_{j=1}^{S_{l+1}} \frac{\partial J}{\partial z_j^{(l+1)}} \frac{\partial z_j^{(l+1)}}{\partial z^{(l)}} = \sum_{j=1}^{S_{l+1}} \delta_j^{(l+1)} \frac{\partial W_j^{(l+1)} f(z^{(l)})}{\partial z^{(l)}}$$

$$= \sum_{j=1}^{S_{l+1}} \delta_j^{(l+1)} W_j^{(l+1)} \bullet f'(z^{(l)}) = (W^{(l+1)})^T \delta^{(l+1)} \bullet f'(z^{(l)})$$

Unlike the formulation in the previous slide, here, we let W(l) be the weights between the layer l-1 and l.



Mathematics of back propagation in CNN (3)

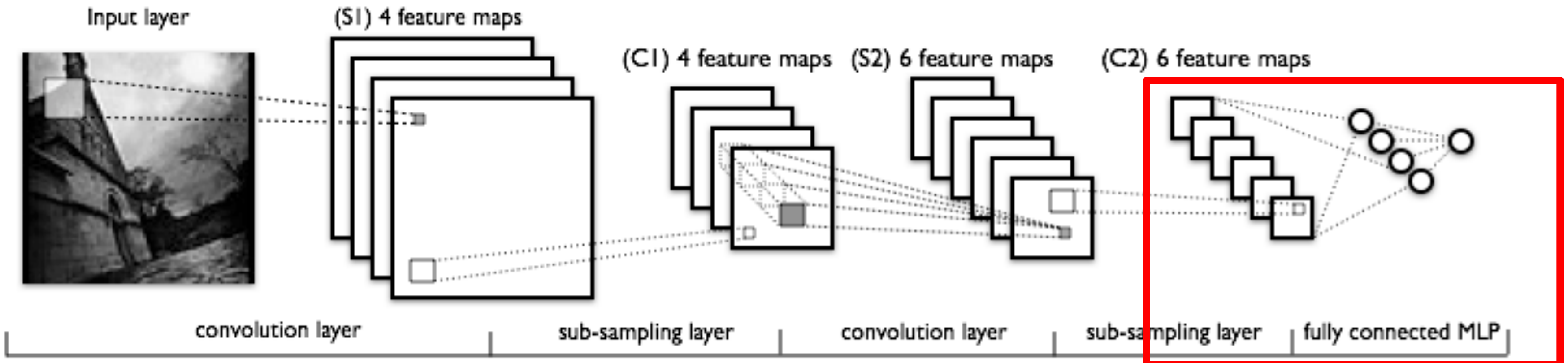
The gradient w.r.t. W and b:

$$\frac{\partial J}{\partial W^{(l)}} = \sum_{i=1}^{S_l} \frac{\partial J}{\partial z_i^{(l)}} \frac{\partial z_i^{(l)}}{\partial W^{(l)}} = \sum_{i=1}^{S_l} \delta_i^l \begin{pmatrix} \ldots & \frac{\partial W_i a^{(l-1)}}{\partial W_j^{(l)}} & \ldots \end{pmatrix}$$

$$= \sum_{i=1}^{S_l} \delta_i^l \begin{pmatrix} \ldots 0 \ldots & a^{(l-1)} & \ldots 0 \ldots \end{pmatrix}$$

$$= a^{(l-1)} \begin{pmatrix} \delta_1^l & \ldots & \delta_i^l & \ldots \end{pmatrix} = a^{(l-1)} (\delta^l)^T$$

$$\frac{\partial J}{\partial b^{(l)}} = \frac{\partial J}{\partial z^{(l)}} \frac{\partial (W^{(l)} a^{(l-1)} + b^{(l)})}{\partial b^{(l)}} = \delta^l$$

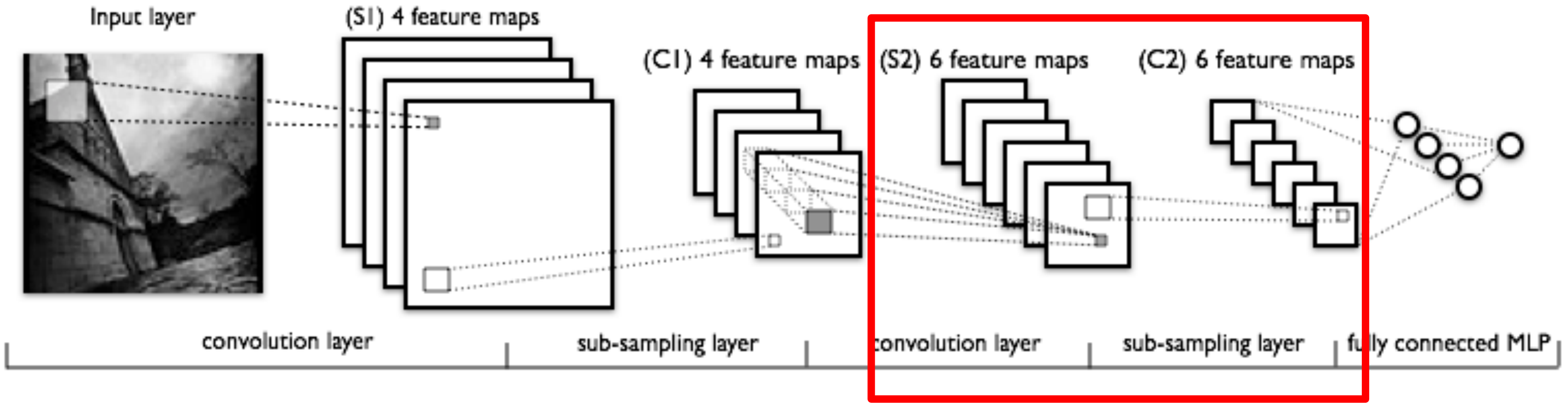


When a convolution layer l followed by a pooling layer (l+1) (4)

$$\delta_j^{(l)} = upsample(\delta_j^{(l+1)}) \bullet f'(z_j^{(l)})$$

Note that here j denote the j-th feature map, but not one element in fully connected layers.

Suppose the activation function is applied after the convolution layer and before the pooling layer. Then, …



When a convolution layer l followed by a pooling layer (l+1) (4)

$$\delta_j^{(l)} = \beta_j^{(l+1)} upsample(\delta_j^{(l+1)}) \bullet f'(z_j^{(l)})$$

Example: suppose the pooling size is 2x2, and the derivative of the error in the (l+1)th layer is

1	3
2	4

For mean-pooling, the up-sampling should be

1	1	3	3
1	1	3	3
2	2	4	4
2	2	4	4

$$(z_j^{(l+1)})_{uv} = \frac{1}{n_{u,v}} \sum_{u,v} (f(z_j^{(l)}))_{u,v}$$

When a convolution layer l followed by a pooling layer (l+1) (4)

$$\delta_j^{(l)} = \beta_j^{(l+1)} upsample(\delta_j^{(l+1)}) \bullet f'(z_j^{(l)})$$

Example: suppose the pooling size is 2x2, and the derivative of the error in the (l+1)th layer is

1	3
2	4

$$(z_j^{(l+1)})_{uv} = \frac{1}{n_{u,v}} \sum_{u,v} (f(z_j^{(l)}))_{u,v}$$

For mean-pooling, the up-sampling should be

1	1	3	3
1	1	3	3
2	2	4	4
2	2	4	4

0.25	0.25	0.75	0.75
0.25	0.25	0.75	0.75
0.5	0.5	1	1
0.5	0.5	1	1

When a convolution layer l followed by a pooling layer (l+1) (4)

$$\delta_j^{(l)} = \beta_j^{(l+1)} upsample(\delta_j^{(l+1)}) \bullet f'(z_j^{(l)})$$

Example: suppose the pooling size is 2x2, and the derivative of the error in the (l+1)th layer is

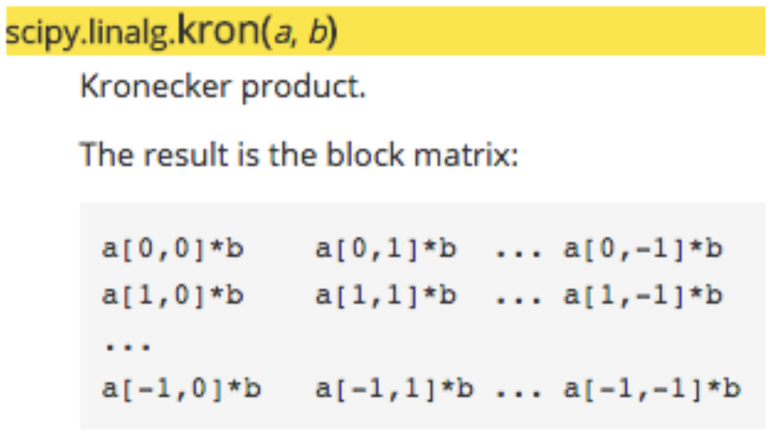
1	3
2	4

For mean-pooling, the up-sampling should be

1	1	3	3
1	1	3	3
2	2	4	4
2	2	4	4

Upsampling operation can be achieved by Kronecker

$$\mathrm{up}(\mathbf{x}) \equiv \mathbf{x} \otimes 1_{n \times n}.$$



scipy.linalg.kron(*a*, *b*)

Kronecker product.

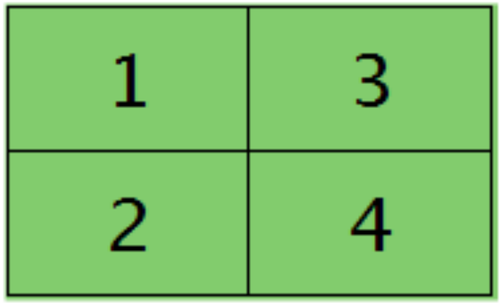
The result is the block matrix:

```
a[0,0]*b    a[0,1]*b  ... a[0,-1]*b
a[1,0]*b    a[1,1]*b  ... a[1,-1]*b
...
a[-1,0]*b   a[-1,1]*b ... a[-1,-1]*b
```

When a convolution layer l followed by a pooling layer (l+1) (4)

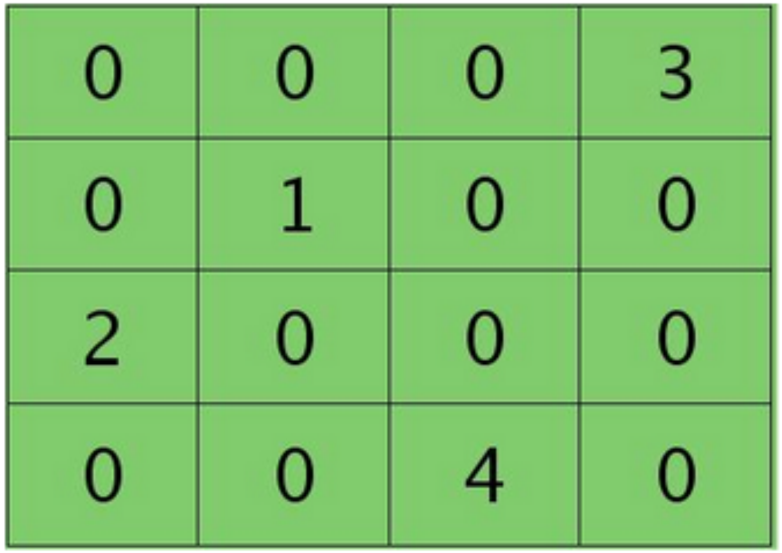
$$\delta_j^{(l)} = \beta_j^{(l+1)} upsample(\delta_j^{(l+1)}) \bullet f'(z_j^{(l)})$$

Example: suppose the pooling size is 2x2, and the sensitivity in the (l+1)th layer is



1	3
2	4

For max-pooling, the up-sampling should be



0	0	0	3
0	1	0	0
2	0	0	0
0	0	4	0

In max pooling the unit which was chosen as the max receives all the error since very small changes in input would perturb the result only through that unit.

When a convolution layer l followed by a pooling layer (l+1) (4)

$$\delta_j^{(l)} = \beta_j^{(l+1)} upsample(\delta_j^{(l+1)}) \bullet f'(z_j^{(l)})$$

Note that, $z_j^{(l)} = \sum_{i \in M^{(l-1)}} a_i^{(l-1)} \circ K_{ij}^{(l)} + b_j^{(l)}, where$ $\circ$ denote cross correlation

The gradient w.r.t. K and b:

$$\frac{\partial J}{\partial b_j^{(l)}} = \sum_{u,v} \frac{\partial J}{\partial (z_j^{(l)})_{u,v}} \frac{\partial (z_j^{(l)})_{u,v}}{\partial b_j^{(l)}} = \sum_{u,v} (\delta_j^{(l)})_{u,v}$$

$$\frac{\partial J}{\partial K_{ij}^{(l)}} = \sum_{u,v} \frac{\partial J}{\partial (z_j^{(l)})_{u,v}} \frac{\partial (z_j^{(l)})_{u,v}}{\partial K_{ij}^{(l)}} = \sum_{u,v} (\delta_j^{(l)})_{u,v} (a_i^{(l-1)})_{u,v},$$

where $(a_i^{(l-1)})_{u,v}$ is the patch in $a_i^{(l-1)}$ that was multiplied elementwise by $K_{ij}^{(l)}$during convolution in order to compute $(z_j^l)u,v$

When a convolution layer l followed by a pooling layer (l+1) (4)

$$\frac{\partial J}{\partial K_{ij}^{(l)}} = \sum_{u,v} \frac{\partial J}{\partial (z_j^{(l)})_{u,v}} \frac{\partial (z_j^{(l)})_{u,v}}{\partial K_{ij}^{(l)}} = \sum_{u,v} (\delta_j^{(l)})_{u,v} (a_i^{(l-1)})_{u,v},$$

where $(a_i^{(l-1)})_{u,v}$ is the patch in $a_i^{(l-1)}$ that was multiplied elementwise by $K_{ij}^{(l)}$during convolution in order to compute $(z_j^l)_{u,v}$

$\delta_j^{(l)}$

0.8	0.1	-0.6
0.3	0.5	0.7
-0.4	0	-0.2

$a_i^{(l-1)}$

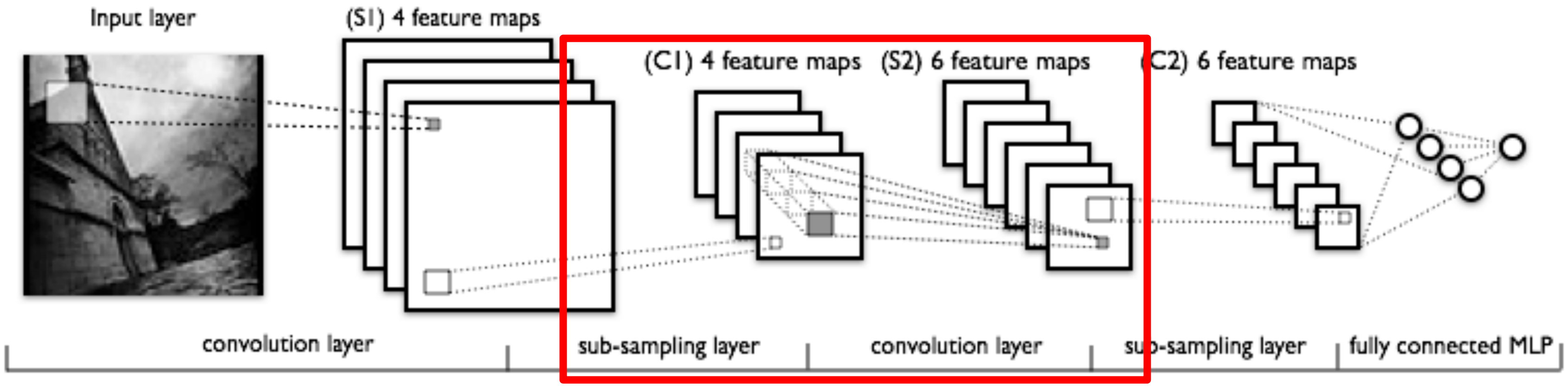
16	2	3	13
5	11	10	8
9	7	6	12
4	14	15	1

$$\frac{\partial J}{\partial K_{ij}^{(l)}} = \sum_{u,v} (\delta_j^{(l)})_{u,v} (a_i^{(l-1)})_{u,v} = a_i^{(l-1)} \circ \delta_j^{(l)}$$

$$= a_i^{(l-1)} * rot90(\delta_j^{(l)}, 2)$$

'valid': without padding 0

When a pooling layer *l* followed by a convolution layer (*l+1*), how to compute the sensitivity of a unit in the layer *l* (5)



$$\delta_i^{(l)} = \sum_{j=1}^{M_j^{(l+1)}} \delta_j^{(l+1)} * K_{ij}^{(l+1)}$$

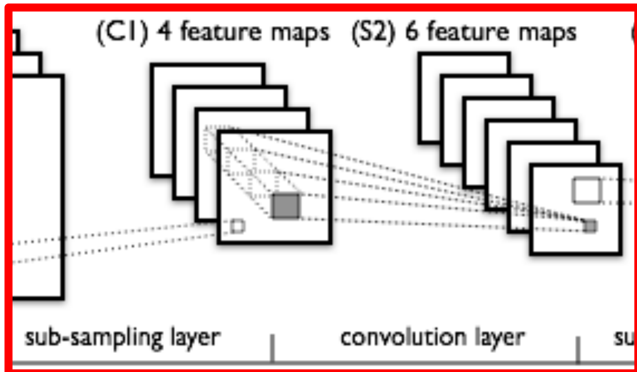
When a pooling layer *l* followed by a convolution layer (*l+1*), how to compute the sensitivity of a unit in the layer *l* (5)

$$\delta_i^{(l)} = \frac{\partial J}{\partial z_i^{(l)}} = \sum_{j \in M^{(l+1)}} \sum_{(z_j^{(l+1)})_{u,v}} (\delta_j^{(l+1)})_{u,v} \frac{\partial (z_j^{(l+1)})_{u,v}}{\partial z_i^{(l)}}$$

$$= \sum_{j \in M^{(l+1)}} \sum_{(z_j^{(l+1)})_{u,v}} (\delta_j^{(l+1)})_{u,v} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & (K_{ij}^{(l+1)})_{(P_i^{(l)})_{u,v}} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$= \sum_{j \in M^{(l+1)}} \delta_j^{(l+1)} * K_{ij}^{(l+1)}$$

where $(P_i^{(l)})_{u,v}$ denotes the patch position in $z_j^{(l)}$ generating the value of $(z_j^{(l+1)})_{u,v}$



When a pooling layer *l* followed by a convolution layer (*l+1*), how to compute the sensitivity of a unit in the layer *l* (5)

Suppose $\delta_j^{(l+1)} = \begin{bmatrix} \delta_{11} & \delta_{12} \\ \delta_{21} & \delta_{22} \end{bmatrix}$ corresponding to $z_j^{(l+1)} = \begin{bmatrix} z_{11} & z_{12} \\ z_{21} & z_{22} \end{bmatrix}$,

and kernel $K_{ij}^{(l+1)} = \begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix}$, then

$$\delta_i^{(l)} = \frac{\partial J}{\partial z_i^{(l)}} = \sum_{j \in M^{(l+1)}} \delta_{11} \begin{bmatrix} k_{11} & k_{12} & 0 \\ k_{21} & k_{22} & 0 \\ 0 & 0 & 0 \end{bmatrix} + \delta_{12} \begin{bmatrix} 0 & k_{11} & k_{12} \\ 0 & k_{21} & k_{22} \\ 0 & 0 & 0 \end{bmatrix} + \delta_{21} \begin{bmatrix} 0 & 0 & 0 \\ k_{11} & k_{12} & 0 \\ k_{21} & k_{22} & 0 \end{bmatrix}$$

$$+\delta_{22} \begin{bmatrix} 0 & 0 & 0 \\ 0 & k_{11} & k_{12} \\ 0 & k_{21} & k_{22} \end{bmatrix} = \sum_{j \in M^{(l+1)}} \begin{bmatrix} \delta_{11} & \delta_{12} \\ \delta_{21} & \delta_{22} \end{bmatrix} * \begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix} = \sum_{j \in M^{(l+1)}} \delta_j^{(l+1)} * K_{ij}^{(l+1)}$$

Real 2D convolution with zero-padding

When a pooling layer *l* followed by a convolution layer (*l+1*), how to compute the sensitivity of a unit in the layer *l* (5)

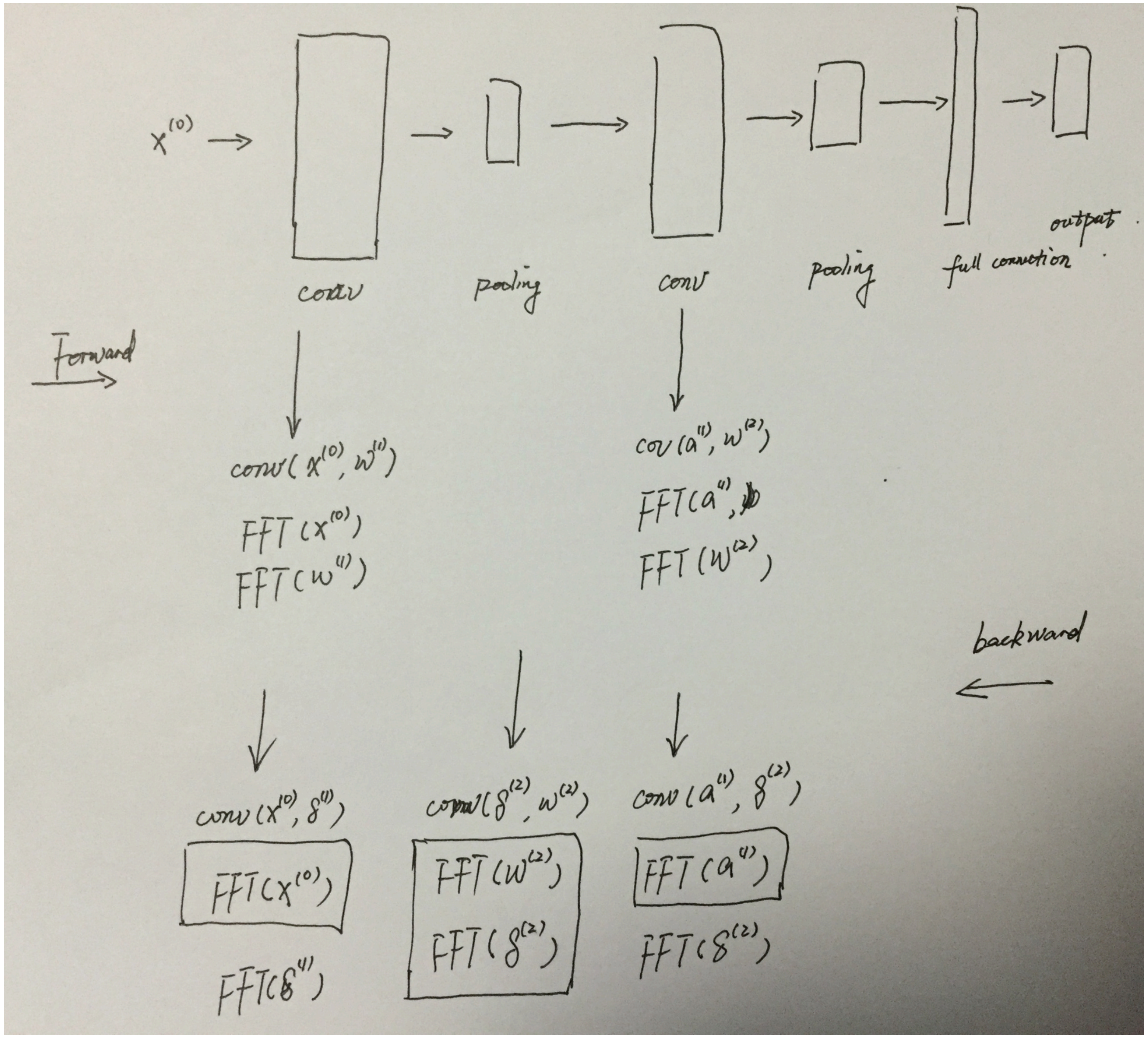
Suppose $\delta_j^{(l+1)} = \begin{bmatrix} \delta_{11} & \delta_{12} \\ \delta_{21} & \delta_{22} \end{bmatrix}$ corresponding to $z_j^{(l+1)}$, and kernel $K_{ij}^{(l+1)} = \begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix}$, then

$$\delta_i^{(l)} = \frac{\partial J}{\partial z_i^{(l)}} = \sum_{j \in M^{(l+1)}} \begin{bmatrix} \delta_{11}k_{11} & \delta_{11}k_{12} + \delta_{12}k_{11} & \delta_{12}k_{12} \\ \delta_{11}k_{21} + \delta_{21}k_{11} & \delta_{11}k_{22} + \delta_{12}k_{21} + \delta_{21}k_{12} + \delta_{22}k_{11} & \delta_{12}k_{22} + \delta_{22}k_{12} \\ \delta_{21}k_{21} & \delta_{21}k_{22} + \delta_{22}k_{21} & \delta_{22}k_{22} \end{bmatrix}$$

$$\delta_j^{(l+1)} * K_{ij}^{(l+1)} = \begin{bmatrix} \delta_{11} & \delta_{12} \\ \delta_{21} & \delta_{22} \end{bmatrix} * \begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & \delta_{11} & \delta_{12} & 0 \\ 0 & \delta_{21} & \delta_{22} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \circ \begin{bmatrix} k_{22} & k_{21} \\ k_{12} & k_{11} \end{bmatrix}$$

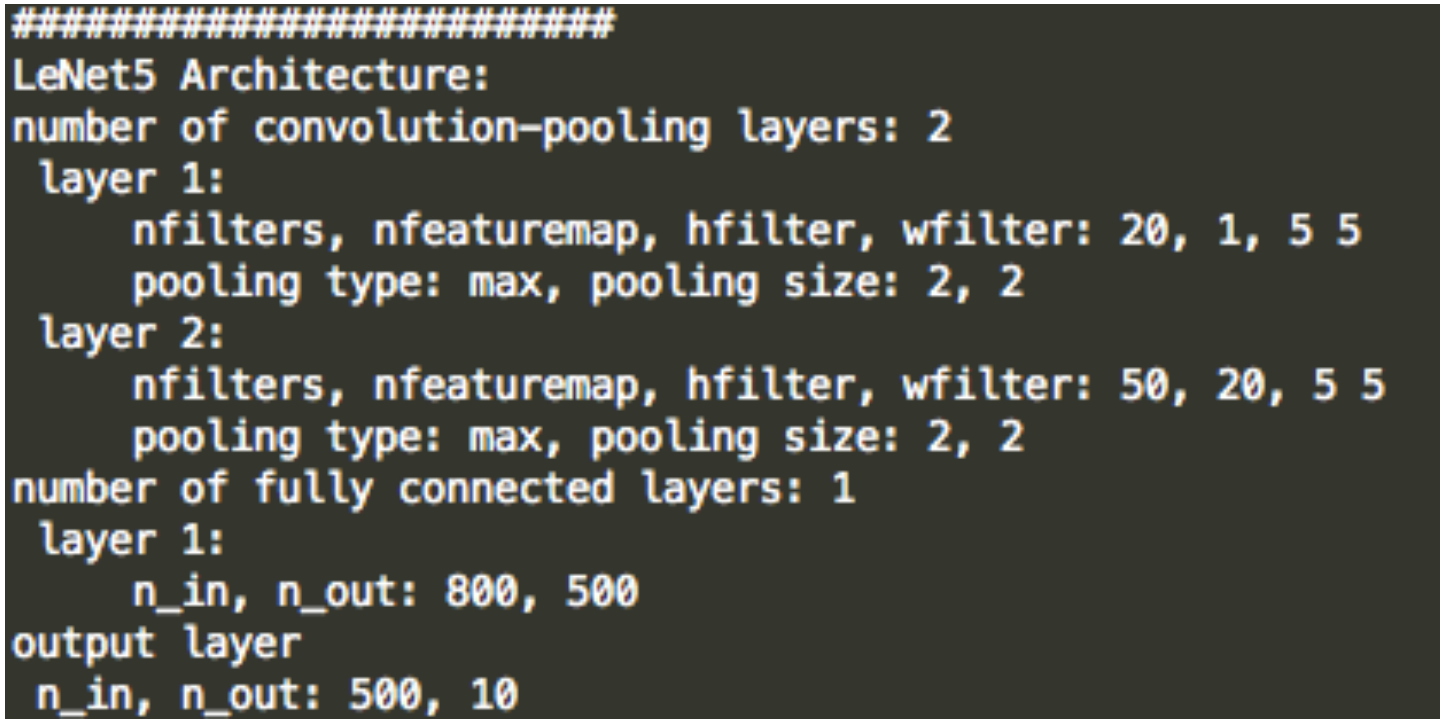
$$= [\begin{matrix} \delta_{11}k_{11} & \delta_{11}k_{12} + \delta_{12}k_{11} & \delta_{12}k_{12} \\ \delta_{11}k_{21} + \delta_{21}k_{11} & \delta_{11}k_{22} + \delta_{12}k_{21} + \delta_{21}k_{12} + \delta_{22}k_{11} & \delta_{12}k_{22} + \delta_{22}k_{12} \\ \delta_{21}k_{21} & \delta_{21}k_{22} + \delta_{22}k_{21} & \delta_{22}k_{22} \end{matrix}]$$

Summary



$x^{(0)}$
conv
pooling
conv
pooling
full connection
output
Forward
backward
$conv(x^{(0)}, w^{(1)})$
$FFT(x^{(0)})$
$FFT(w^{(1)})$
$cov(a^{(1)}, w^{(2)})$
$FFT(a^{(1)})$
$FFT(w^{(2)})$
$conv(x^{(0)}, \delta^{(1)})$
$FFT(x^{(0)})$
$FFT(\delta^{(1)})$
$conv(\delta^{(2)}, w^{(2)})$
$FFT(w^{(2)})$
$FFT(\delta^{(2)})$
$conv(a^{(1)}, \delta^{(2)})$
$FFT(a^{(1)})$
$FFT(\delta^{(2)})$

Practice on MNIST dataset



```
###############################
LeNet5 Architecture:
number of convolution-pooling layers: 2
 layer 1:
     nfilters, nfeaturemap, hfilter, wfilter: 20, 1, 5 5
     pooling type: max, pooling size: 2, 2
 layer 2:
     nfilters, nfeaturemap, hfilter, wfilter: 50, 20, 5 5
     pooling type: max, pooling size: 2, 2
number of fully connected layers: 1
 layer 1:
     n_in, n_out: 800, 500
output layer
 n_in, n_out: 500, 10
```

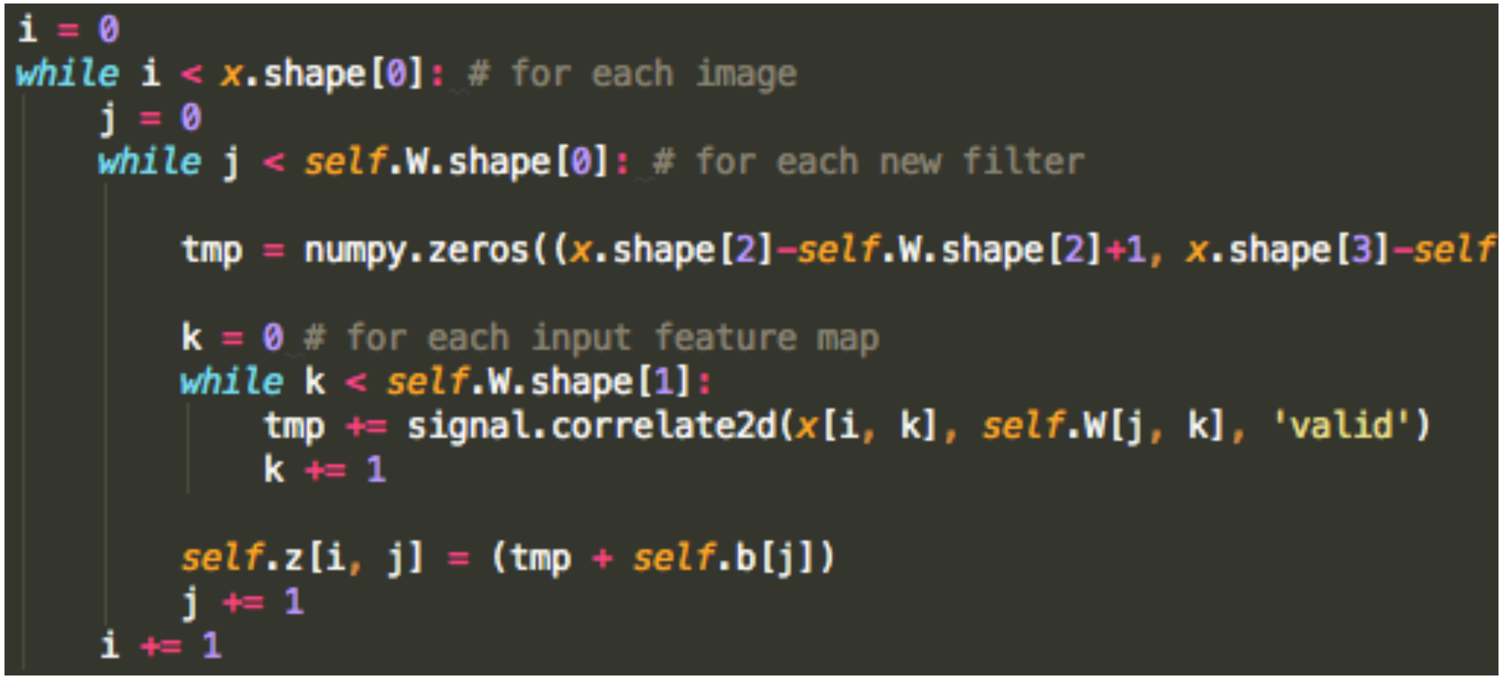
nkerns=[20, 50], kern_size=[[5, 5], [5, 5]], pool_size=[2, 2], n_list_full_conn_layer_nodes=[500], n_out=10, learning_rate=0.1, n_epochs=200, batch_size=500

Bottle neck of computational efficiency (1)

Feedforward.convolution

- for loop:

(500, 1, 28, 28)=>(500, 20, 24, 24)=>(500, 20, 12, 12)=>(500, 50, 8, 8)=>(500, 50, 4, 4)

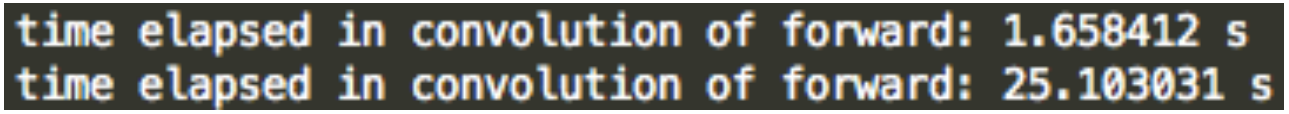


```
i = 0
while i < x.shape[0]: # for each image
    j = 0
    while j < self.W.shape[0]: # for each new filter

        tmp = numpy.zeros((x.shape[2]-self.W.shape[2]+1, x.shape[3]-self

        k = 0 # for each input feature map
        while k < self.W.shape[1]:
            tmp += signal.correlate2d(x[i, k], self.W[j, k], 'valid')
            k += 1

        self.z[i, j] = (tmp + self.b[j])
        j += 1
    i += 1
```



```
time elapsed in convolution of forward: 1.658412 s
time elapsed in convolution of forward: 25.103031 s
```

one epoch needs 100x …

Bottle neck of computational efficiency (1)

Feedforward.convolution

(500, 1, 28, 28)=>(500, 20, 24, 24)=>(500, 20, 12, 12)=>(500, 50, 8, 8)=>(500, 50, 4, 4)

Fast convolution ?

- Vectorization

  On Vectorization of Deep Convolutional Neural Networks for Vision Tasks, AAAI, 2015.

- Convolution by FFT

  Even faster convolutions in Theano using FFTs

  http://benanne.github.io/2014/05/12/fft-convolutions-in-theano.html

- GPU

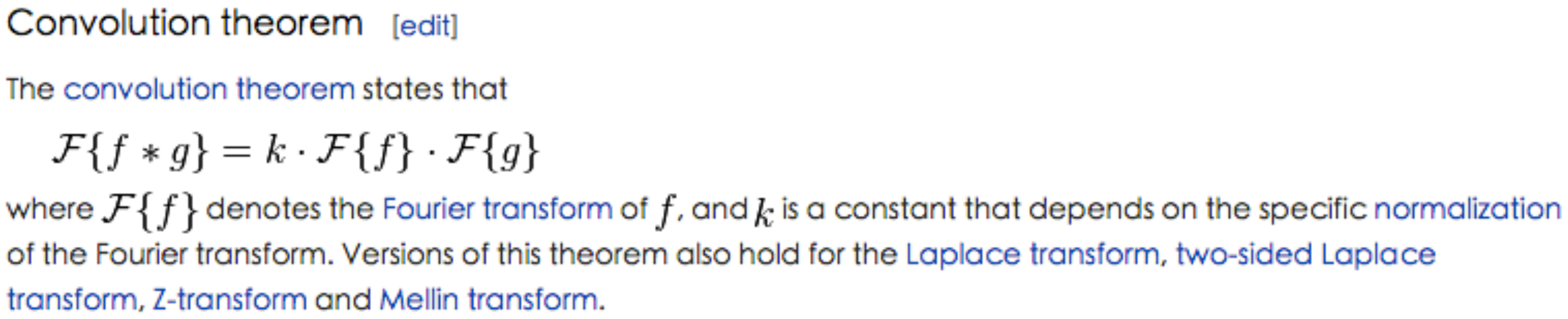
  many many works!

Bottle neck of computational efficiency (1)

Feedforward.convolution

- Convolution by fast Fourier transform (FFT)

(500, 1, 28, 28)=>(500, 20, 24, 24)=>(500, 20, 12, 12)=>(500, 50, 8, 8)=>(500, 50, 4, 4)

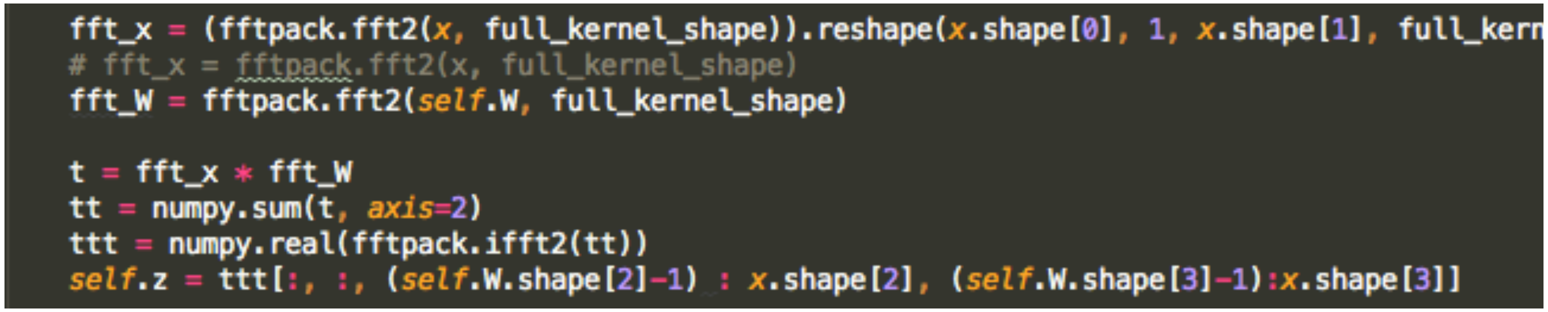


Convolution theorem [edit]

The convolution theorem states that

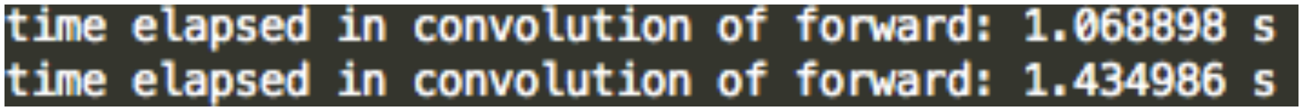
$$\mathcal{F}\{f * g\} = k \cdot \mathcal{F}\{f\} \cdot \mathcal{F}\{g\}$$

where $\mathcal{F}\{f\}$ denotes the Fourier transform of f, and k is a constant that depends on the specific normalization of the Fourier transform. Versions of this theorem also hold for the Laplace transform, two-sided Laplace transform, Z-transform and Mellin transform.



```
fft_x = (fftpack.fft2(x, full_kernel_shape)).reshape(x.shape[0], 1, x.shape[1], full_kern
# fft_x = fftpack.fft2(x, full_kernel_shape)
fft_W = fftpack.fft2(self.W, full_kernel_shape)

t = fft_x * fft_W
tt = numpy.sum(t, axis=2)
ttt = numpy.real(fftpack.ifft2(tt))
self.z = ttt[:, :, (self.W.shape[2]-1) : x.shape[2], (self.W.shape[3]-1):x.shape[3]]
```



```
time elapsed in convolution of forward: 1.068898 s
time elapsed in convolution of forward: 1.434986 s
```

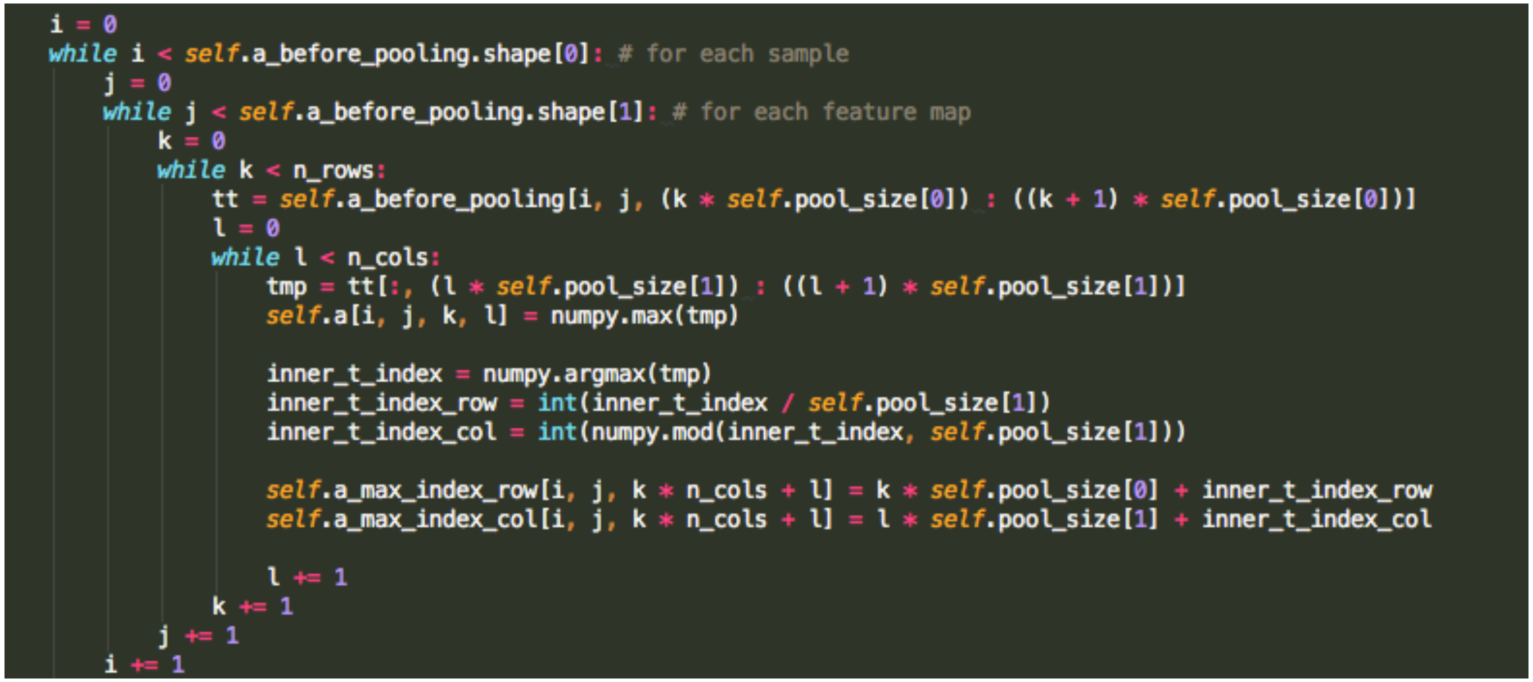
one epoch needs 100x …

Bottle neck of computational efficiency (2)

Feedforward.pooling

- For loop

(500, 1, 28, 28)=>(500, 20, 24, 24)=>(500, 20, 12, 12)=>(500, 50, 8, 8)=>(500, 50, 4, 4)

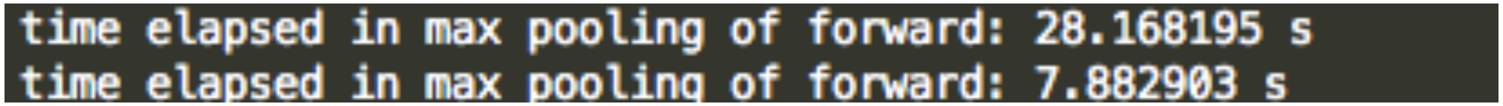


```
i = 0
while i < self.a_before_pooling.shape[0]: # for each sample
    j = 0
    while j < self.a_before_pooling.shape[1]: # for each feature map
        k = 0
        while k < n_rows:
            tt = self.a_before_pooling[i, j, (k * self.pool_size[0]) : ((k + 1) * self.pool_size[0])]
            l = 0
            while l < n_cols:
                tmp = tt[:, (l * self.pool_size[1]) : ((l + 1) * self.pool_size[1])]
                self.a[i, j, k, l] = numpy.max(tmp)

                inner_t_index = numpy.argmax(tmp)
                inner_t_index_row = int(inner_t_index / self.pool_size[1])
                inner_t_index_col = int(numpy.mod(inner_t_index, self.pool_size[1]))

                self.a_max_index_row[i, j, k * n_cols + l] = k * self.pool_size[0] + inner_t_index_row
                self.a_max_index_col[i, j, k * n_cols + l] = l * self.pool_size[1] + inner_t_index_col

                l += 1
            k += 1
        j += 1
    i += 1
```



```
time elapsed in max pooling of forward: 28.168195 s
time elapsed in max pooling of forward: 7.882903 s
```

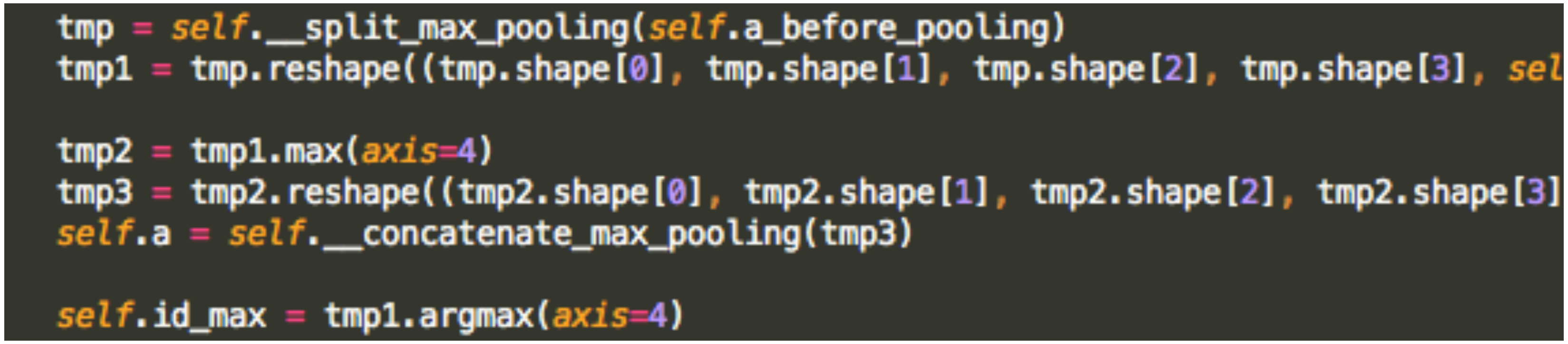
one epoch needs 100x …

Bottle neck of computational efficiency (2)

Feedforward.pooling

- Nice 'split' and 'concatenate'

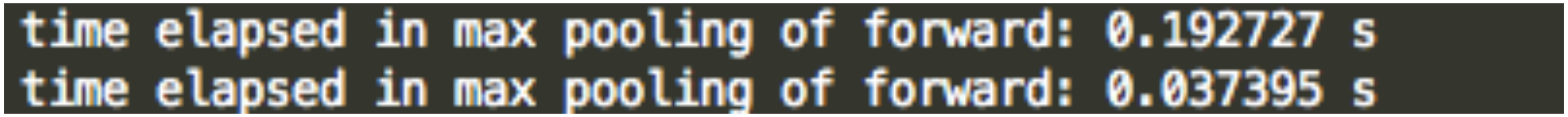
(500, 1, 28, 28)=>(500, 20, 24, 24)=>(500, 20, 12, 12)=>(500, 50, 8, 8)=>(500, 50, 4, 4)



```
    tmp = self.__split_max_pooling(self.a_before_pooling)
    tmp1 = tmp.reshape((tmp.shape[0], tmp.shape[1], tmp.shape[2], tmp.shape[3], sel

    tmp2 = tmp1.max(axis=4)
    tmp3 = tmp2.reshape((tmp2.shape[0], tmp2.shape[1], tmp2.shape[2], tmp2.shape[3]
    self.a = self.__concatenate_max_pooling(tmp3)

    self.id_max = tmp1.argmax(axis=4)
```



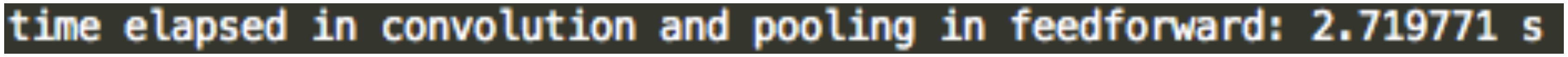
```
time elapsed in max pooling of forward: 0.192727 s
time elapsed in max pooling of forward: 0.037395 s
```

one epoch needs 100x …

Bottle neck of computational efficiency (3)

Time cost in Feedforward

(500, 1, 28, 28)=>(500, 20, 24, 24)=>(500, 20, 12, 12)=>(500, 50, 8, 8)=>(500, 50, 4, 4)



```
time elapsed in convolution and pooling in feedforward: 2.719771 s
```

one epoch needs 100x …

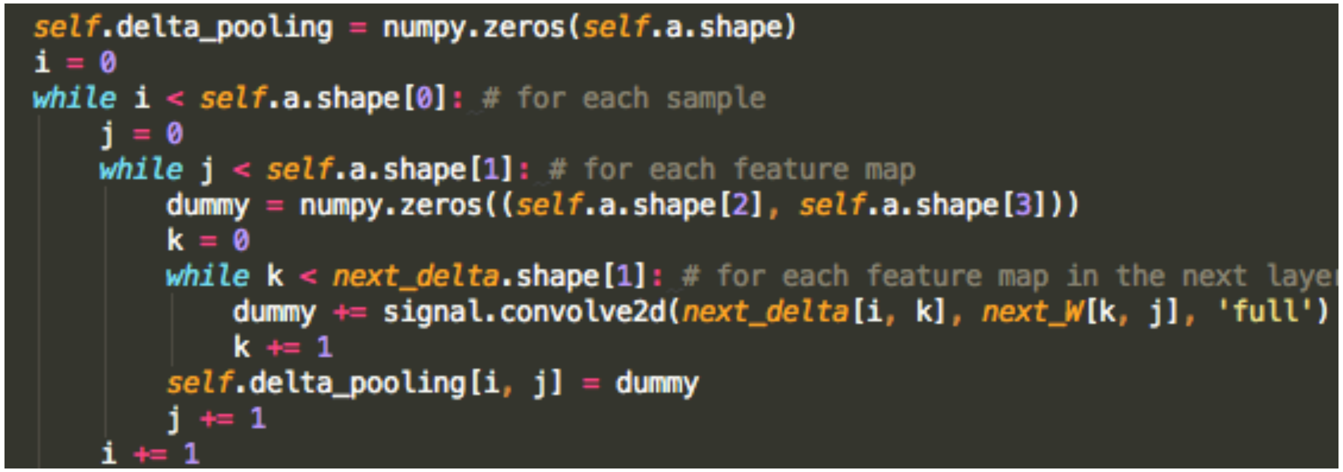
Bottle neck of computational efficiency (4)

backward.compute_poolLayer_delta_from_convLayer

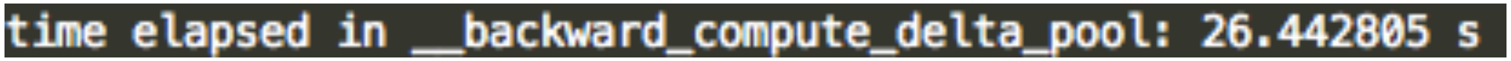
- for loop

(500, 1, 28, 28)=>(500, 20, 24, 24)=>(500, 20, 12, 12)=>(500, 50, 8, 8)=>(500, 50, 4, 4)

$$\delta_i^{(l)} = \sum_{j=1}^{M_j^{(l+1)}} \delta_j^{(l+1)} * K_{ij}^{(l+1)}$$



```
self.delta_pooling = numpy.zeros(self.a.shape)
i = 0
while i < self.a.shape[0]: # for each sample
    j = 0
    while j < self.a.shape[1]: # for each feature map
        dummy = numpy.zeros((self.a.shape[2], self.a.shape[3]))
        k = 0
        while k < next_delta.shape[1]: # for each feature map in the next laye
            dummy += signal.convolve2d(next_delta[i, k], next_W[k, j], 'full')
            k += 1
        self.delta_pooling[i, j] = dummy
        j += 1
    i += 1
```



```
time elapsed in __backward_compute_delta_pool: 26.442805 s
```

one epoch needs 100x …

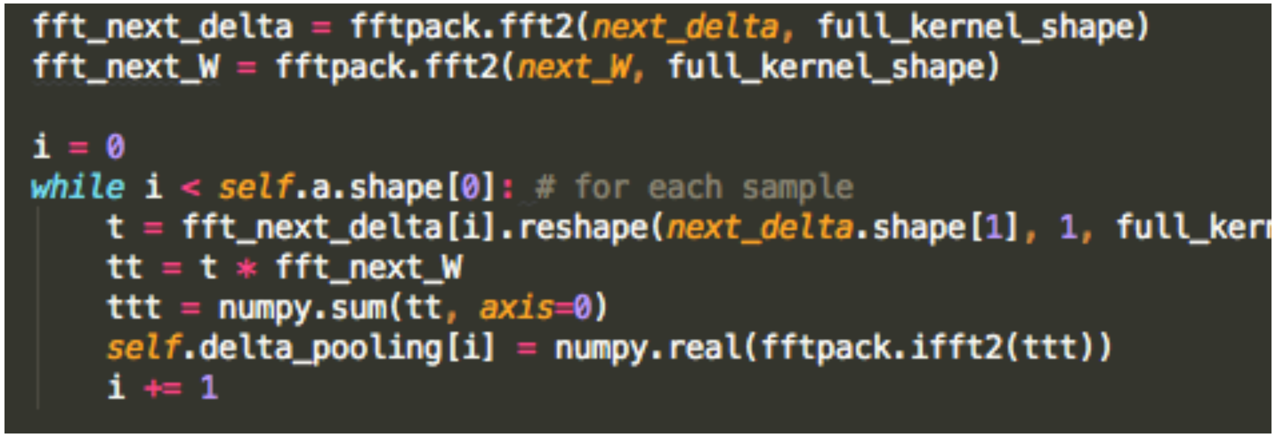
Bottle neck of computational efficiency (4)

backward.compute_poolLayer_delta_from_convLayer

- FFT

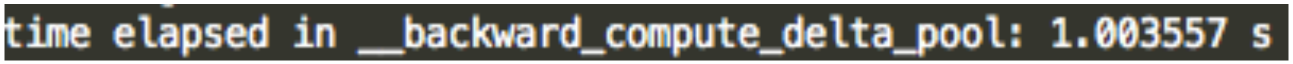
(500, 1, 28, 28)=>(500, 20, 24, 24)=>(500, 20, 12, 12)=>(500, 50, 8, 8)=>(500, 50, 4, 4)

$$\delta_i^{(l)} = \sum_{j=1}^{M_j^{(l+1)}} \delta_j^{(l+1)} * K_{ij}^{(l+1)}$$



```
fft_next_delta = fftpack.fft2(next_delta, full_kernel_shape)
fft_next_W = fftpack.fft2(next_W, full_kernel_shape)

i = 0
while i < self.a.shape[0]: # for each sample
    t = fft_next_delta[i].reshape(next_delta.shape[1], 1, full_ker
    tt = t * fft_next_W
    ttt = numpy.sum(tt, axis=0)
    self.delta_pooling[i] = numpy.real(fftpack.ifft2(ttt))
    i += 1
```



```
time elapsed in __backward_compute_delta_pool: 1.003557 s
```

one epoch needs 100x …

Bottle neck of computational efficiency (5)

backward.update_W_b

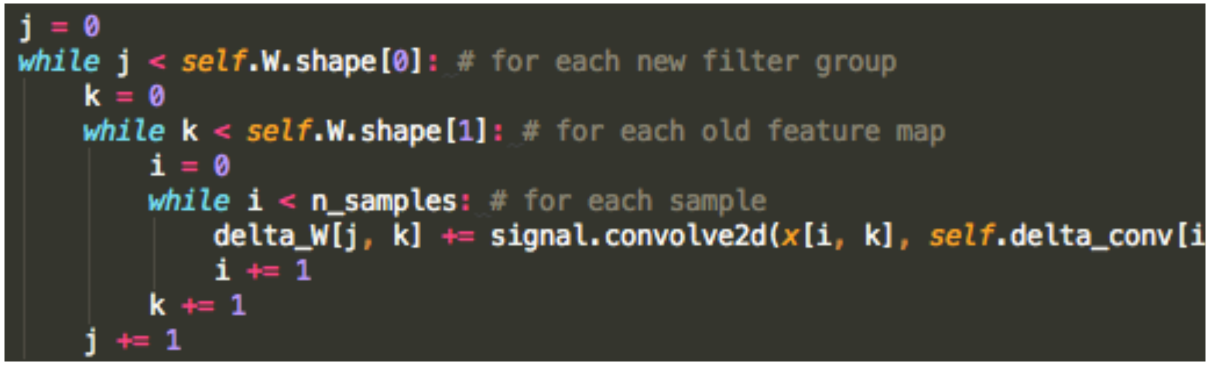
- for loop

(500, 1, 28, 28)=>(500, 20, 24, 24)=>(500, 20, 12, 12)=>(500, 50, 8, 8)=>(500, 50, 4, 4)

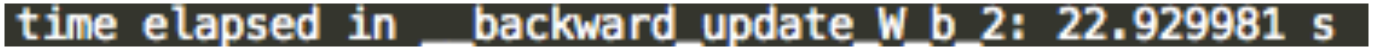
$$\frac{\partial J}{\partial K_{ij}^{(l)}} = \sum_{u,v} (\delta_j^{(l)})_{u,v} (a_i^{(l-1)})_{u,v} = a_i^{(l-1)} \circ \delta_j^{(l)}$$

$$= a_i^{(l-1)} * rot90(\delta_j^{(l)}, 2)$$

'valid': without padding 0



```
j = 0
while j < self.W.shape[0]: # for each new filter group
    k = 0
    while k < self.W.shape[1]: # for each old feature map
        i = 0
        while i < n_samples: # for each sample
            delta_W[j, k] += signal.convolve2d(x[i, k], self.delta_conv[i
            i += 1
        k += 1
    j += 1
```



```
time elapsed in __backward_update_W_b_2: 22.929981 s
```

one epoch needs 100x …

Bottle neck of computational efficiency (5)

backward.update_W_b

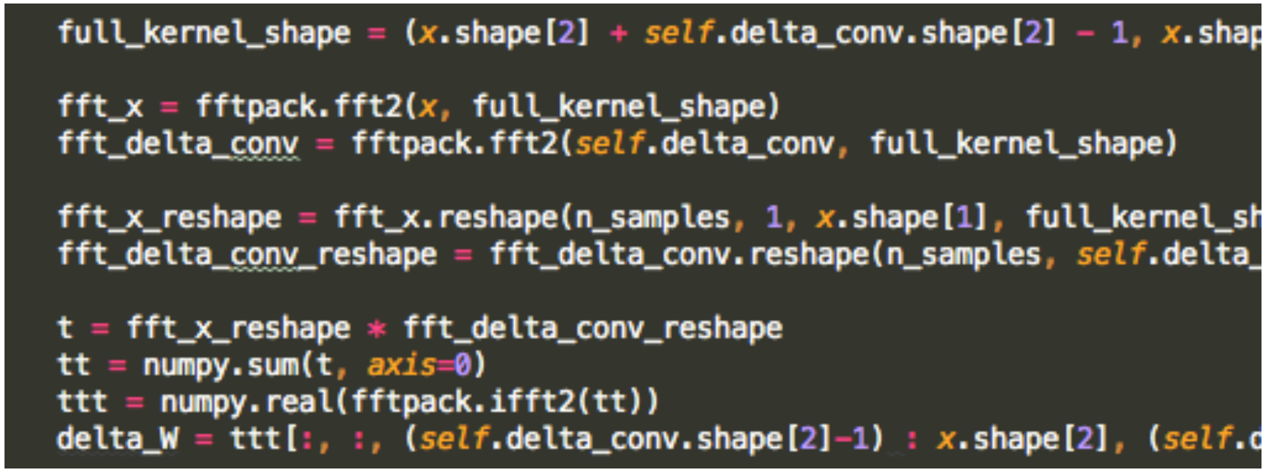
- FFT

(500, 1, 28, 28)=>(500, 20, 24, 24)=>(500, 20, 12, 12)=>(500, 50, 8, 8)=>(500, 50, 4, 4)

$$\frac{\partial J}{\partial K_{ij}^{(l)}} = \sum_{u,v} (\delta_j^{(l)})_{u,v} (a_i^{(l-1)})_{u,v} = a_i^{(l-1)} \circ \delta_j^{(l)}$$

$$= a_i^{(l-1)} * rot90(\delta_j^{(l)}, 2)$$

'valid': without padding 0

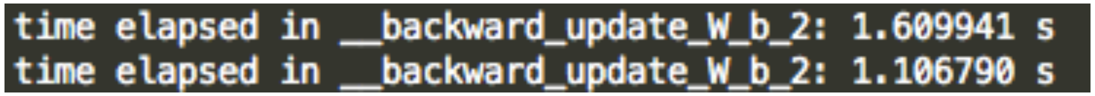


```
full_kernel_shape = (x.shape[2] + self.delta_conv.shape[2] - 1, x.shap

fft_x = fftpack.fft2(x, full_kernel_shape)
fft_delta_conv = fftpack.fft2(self.delta_conv, full_kernel_shape)

fft_x_reshape = fft_x.reshape(n_samples, 1, x.shape[1], full_kernel_sh
fft_delta_conv_reshape = fft_delta_conv.reshape(n_samples, self.delta_

t = fft_x_reshape * fft_delta_conv_reshape
tt = numpy.sum(t, axis=0)
ttt = numpy.real(fftpack.ifft2(tt))
delta_W = ttt[:, :, (self.delta_conv.shape[2]-1) : x.shape[2], (self.d
```



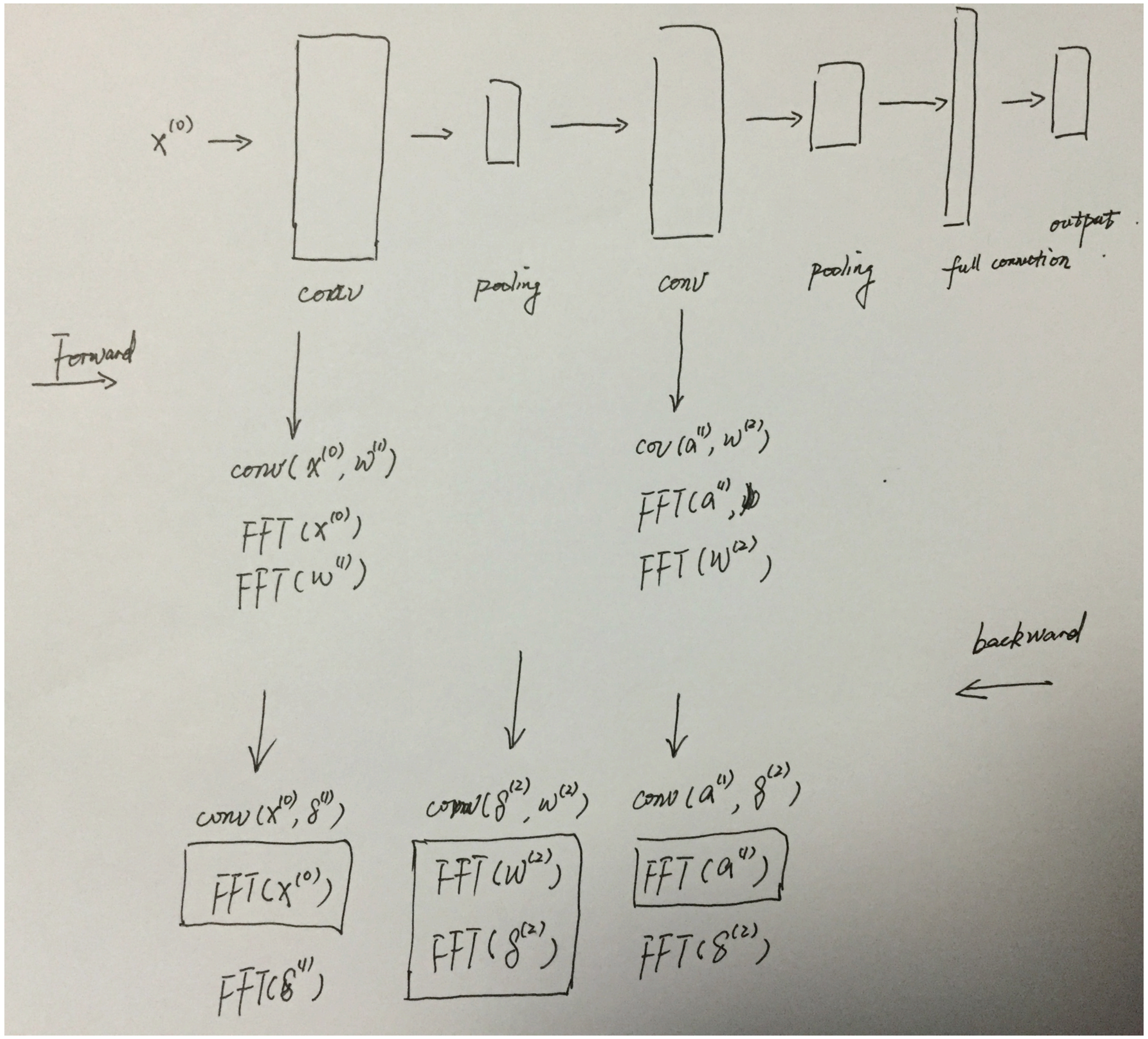
```
time elapsed in __backward_update_W_b_2: 1.609941 s
time elapsed in __backward_update_W_b_2: 1.106790 s
```

one epoch needs 100x …

Bottle neck of computational efficiency (6)

Reuse FFT results

(500, 1, 28, 28)=>(500, 20, 24, 24)=>(500, 20, 12, 12)=>(500, 50, 8, 8)=>(500, 50, 4, 4)

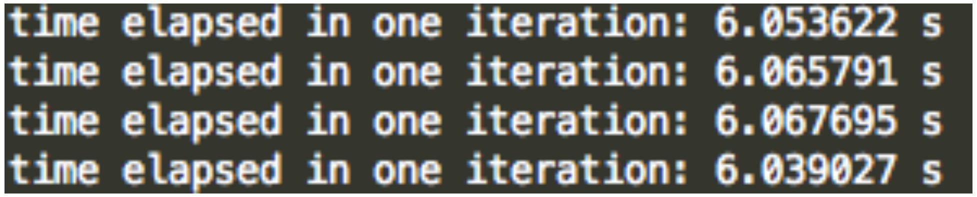


Bottle neck of computational efficiency (6)

Reuse FFT results

(500, 1, 28, 28)=>(500, 20, 24, 24)=>(500, 20, 12, 12)=>(500, 50, 8, 8)=>(500, 50, 4, 4)

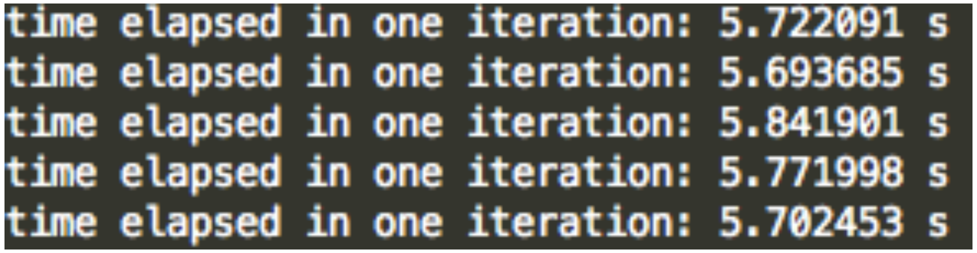
Before:



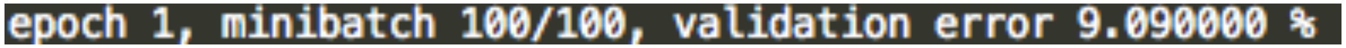
```
time elapsed in one iteration: 6.053622 s
time elapsed in one iteration: 6.065791 s
time elapsed in one iteration: 6.067695 s
time elapsed in one iteration: 6.039027 s
```

About 600 s for one epoch

After:



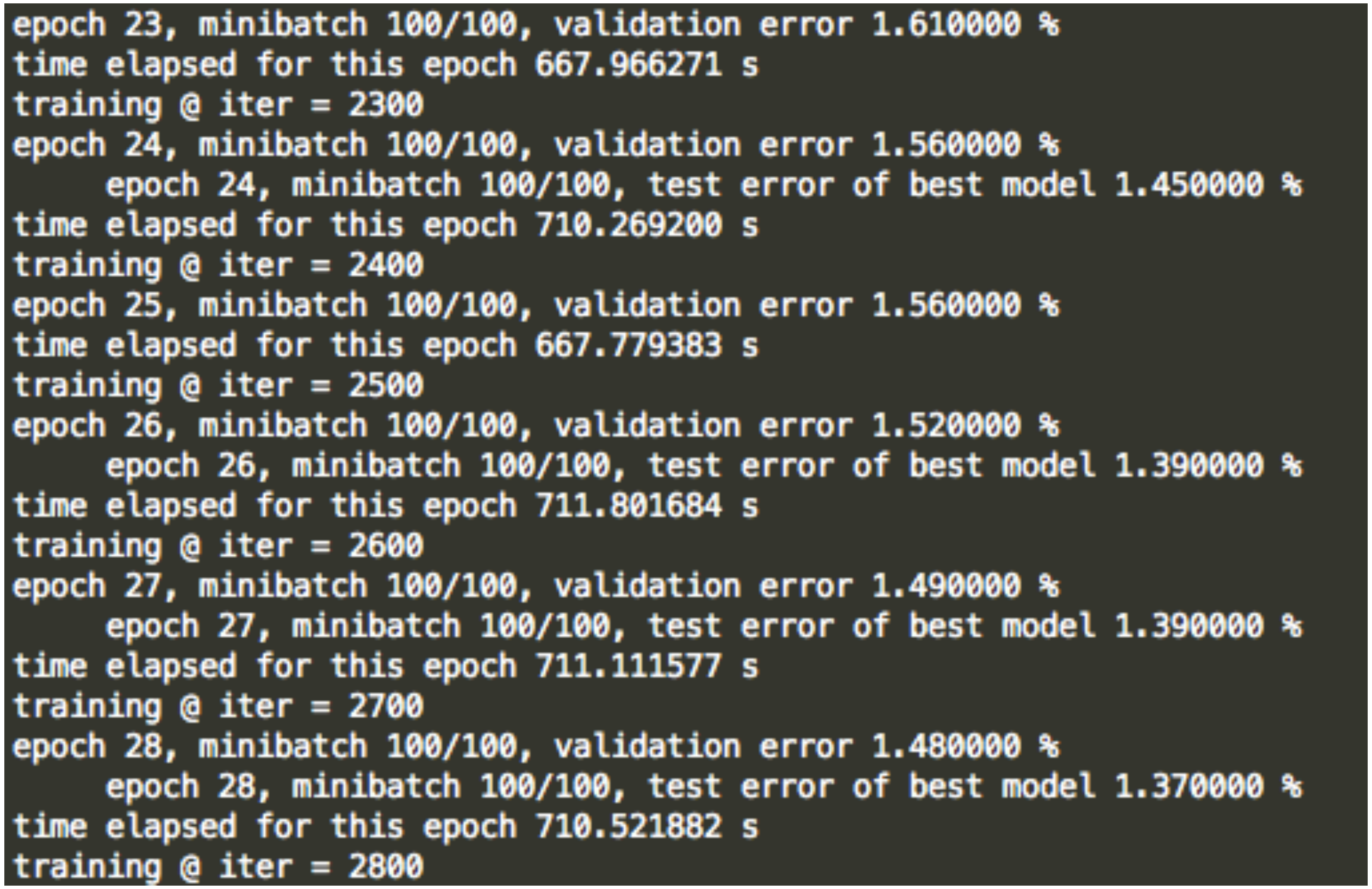
```
time elapsed in one iteration: 5.722091 s
time elapsed in one iteration: 5.693685 s
time elapsed in one iteration: 5.841901 s
time elapsed in one iteration: 5.771998 s
time elapsed in one iteration: 5.702453 s
```



```
epoch 1, minibatch 100/100, validation error 9.090000 %
```

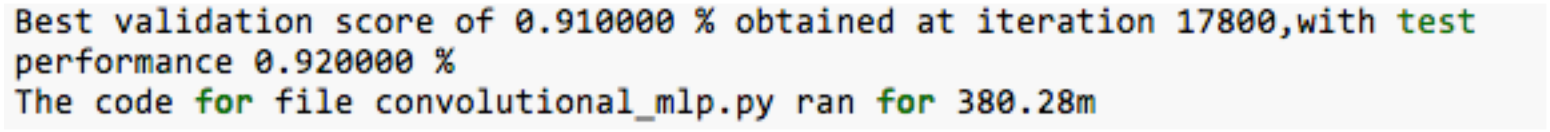
Only a bit faster

Results on MNIST dataset



```
epoch 23, minibatch 100/100, validation error 1.610000 %
time elapsed for this epoch 667.966271 s
training @ iter = 2300
epoch 24, minibatch 100/100, validation error 1.560000 %
     epoch 24, minibatch 100/100, test error of best model 1.450000 %
time elapsed for this epoch 710.269200 s
training @ iter = 2400
epoch 25, minibatch 100/100, validation error 1.560000 %
time elapsed for this epoch 667.779383 s
training @ iter = 2500
epoch 26, minibatch 100/100, validation error 1.520000 %
     epoch 26, minibatch 100/100, test error of best model 1.390000 %
time elapsed for this epoch 711.801684 s
training @ iter = 2600
epoch 27, minibatch 100/100, validation error 1.490000 %
     epoch 27, minibatch 100/100, test error of best model 1.390000 %
time elapsed for this epoch 711.111577 s
training @ iter = 2700
epoch 28, minibatch 100/100, validation error 1.480000 %
     epoch 28, minibatch 100/100, test error of best model 1.370000 %
time elapsed for this epoch 710.521882 s
training @ iter = 2800
```

Results of LISA cnn:



```
Best validation score of 0.910000 % obtained at iteration 17800,with test
performance 0.920000 %
The code for file convolutional_mlp.py ran for 380.28m
```

Practice on the basketball problem

Basketball Dataset

- We have,
  - ✓ a training/validation set: more than 80,000 negatives and 465 positives for 2-frame samples, and more than 80,000 negatives and 586 positives for 1-frame samples. 80 percentage is treated as the training set and 20 percentage is put to validation set.
  - ✓ a testing set: 158 positives and 28451 negatives for 2-frame samples and similar number of samples for the 1-frame case.

Tips

If you want to try this model on a new dataset, here are a few tips that can help you get better results:

- Whitening the data (e.g. with PCA)
- Decay the learning rate in each epoch

Learning rate

There is a great deal of literature on choosing a good learning rate. The simplest solution is to simply have a constant rate. Rule of thumb: try several log-spaced values $(10^{-1}, 10^{-2}, \ldots)$ and narrow the (logarithmic) grid search to the region where you obtain the lowest validation error.

Decreasing the learning rate over time is sometimes a good idea. One simple rule for doing that is $\frac{\mu_0}{1+d\times t}$ where μ_0 is the initial rate (chosen, perhaps, using the grid search technique explained above), d is a so-called "decrease constant" which controls the rate at which the learning rate decreases (typically, a smaller positive number, 10^{-3} and smaller) and t is the epoch/stage.

Section 4.7 details procedures for choosing a learning rate for each parameter (weight) in our network and for choosing them adaptively based on the error of the classifier.

Weight initialization

At initialization we want the weights to be small enough around the origin so that the activation function operates in its linear regime, where gradients are the largest. Other desirable properties, especially for deep networks, are to conserve variance of the activation as well as variance of back-propagated gradients from layer to layer. This allows information to flow well upward and downward in the network and reduces discrepancies between layers. Under some assumptions, a compromise between these two constraints leads to the following initialization: $uniform[-\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}]$ for tanh and $uniform[-4*\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, 4*\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}]$ for sigmoid. Where fan_{in} is the number of inputs and fan_{out} the number of hidden units. For mathematical considerations please refer to [Xavier10].

Number of hidden units

This hyper-parameter is very much dataset-dependent. Vaguely speaking, the more complicated the input distribution is, the more capacity the network will require to model it, and so the larger the number of hidden units that will be needed (note that the number of weights in a layer, perhaps a more direct measure of capacity, is $D \times D_h$ (recall D is the number of inputs and D_h is the number of hidden units).

Unless we employ some regularization scheme (early stopping or L1/L2 penalties), a typical number of hidden units vs. generalization performance graph will be U-shaped.

Regularization parameter

Typical values to try for the L1/L2 regularization parameter λ are $10^{-2}, 10^{-3}, \ldots$. In the framework that we described so far, optimizing this parameter will not lead to significantly better solutions, but is worth exploring nonetheless.

Number of filters

Since feature map size decreases with depth, layers near the input layer will tend to have fewer filters while layers higher up can have much more. In fact, to equalize computation at each layer, the product of the number of features and the number of pixel positions is typically picked to be roughly constant across layers. To preserve the information about the input would require keeping the total number of activations (number of feature maps times number of pixel positions) to be non-decreasing from one layer to the next (of course we could hope to get away with less when we are doing supervised learning). The number of feature maps directly controls capacity and so that depends on the number of available examples and the complexity of the task.

Filter Shape

Common filter shapes found in the litterature vary greatly, usually based on the dataset. Best results on MNIST-sized images (28x28) are usually in the 5x5 range on the first layer, while natural image datasets (often with hundreds of pixels in each dimension) tend to use larger first-layer filters of shape 12x12 or 15x15.

The trick is thus to find the right level of "granularity" (i.e. filter shapes) in order to create abstractions at the proper scale, given a particular dataset.

Max Pooling Shape

Typical values are 2x2 or no max-pooling. Very large input images may warrant 4x4 pooling in the lower-layers. Keep in mind however, that this will reduce the dimension of the signal by a factor of 16, and may result in throwing away too much information.

Shuffling the Examples

The most important practice is getting a training set as large as possible: we expand the training set by adding a new form of distorted data.

generating additional data using transformations may even improve performance

Results on basketball dataset

```
############################
LeNet5 Architecture:
    number of convolution-pooling layers: 2
        layer 1:
            nfilters, nfeaturemap, hfilter, wfilter: 10, 2, 5 9
            pooling type: max, pooling size: 2, 2
        layer 2:
            nfilters, nfeaturemap, hfilter, wfilter: 30, 10, 5 5
            pooling type: max, pooling size: 2, 2
    number of fully connected layers: 1
        layer 1:
            n_in, n_out: 900, 20
    output layer
        n_in, n_out: 20, 2


############################
learning_rate=0.1, n_epochs=200, batch_size=500
```

Results on basketball dataset

References

UFLDL Tutorial, http://ufldl.stanford.edu/tutorial

LISA Deep Learning Tutorial,
http://deeplearning.net/tutorial/lenet.html#lenet

J. Bouvrie, Notes on Convolutional Neural Networks, 2006

DeepFace: Closing the Gap to Human-Level Performance in Face Verification, CVPR’2014

Yann Lecun, Gradient-Based Learning Applied to Document Recognition, proceedings of IEEE, 1998

http://www.cnblogs.com/tornadomeet/p/3468450.html

Assignments

- Derive every formula by yourself in the slides
- Read the CNN code on deep learning tutorial, run it on MNIST dataset, and rewrite / change it to be theano-free.
- Carry out all experiments I showed on the basketball dataset using convolutional neural network